공학석사학위논문

NVIDIA GPU를 위한 OpenMP Device Construct 변환

Translating OpenMP Device Constructs for NVIDIA GPUs

2020년 8월

서울대학교 대학원
컴퓨터공학부
박 대 영

NVIDIA GPU를 위한 OpenMP Device Construct 변환

Translating OpenMP Device Constructs for NVIDIA GPUs

지도교수 이 재 진

이 논문을 공학석사 학위논문으로 제출함

2020년 06월

서울대학교 대학원

컴퓨터공학부

박 대 영

박대영의 석사학위논문을 인준함

2020년 07월

위 원 장	염 헌 영	(인)
부위원장	이 재 진	(인)
위 원	전 병 곤	(인)

요약

본 논문은 OpenMP 4.5 device construct를 사용해 작성된 C 프로그램을 CUDA 프로그램으로 변환하는 컴파일러와 런타임 시스템으로 구성된 프레임워크를 다룬다. 이를 위해 OpenMP C 프로그램을 CUDA 프로그램으로 변환하는 컴파일러 및 런타임 라이브러리를 제시한다. 먼저, OpenMP와 CUDA의 실행 모델, 메모리 모델 및 동기화 과정을 조망하고, source-level 컴파일러의 디자인과 정확성을 확보하기 위한 방법을 설명한다. 또한, 성능 향상을 위한 동적 구간 트리, 동적 커널 선택, 버디 할당자, UDTE와 같은 런타임 시스템 최적화 기술을 도입한다.

spec-accel 1.2 벤치마크를 이용한 실험 결과 gcc7 대비 6배 이상, mriq를 제외한 경우 2배 이상의 성능을 보여준다. 향후 본 논문의 프레임워크를 바탕으로 추가적인 런타임 최적화 기능 및 컴파일러 최적화 기술을 적용할 수 있을 것으로 기대된다.

주요어 : 이종 컴퓨팅, OpenMP, CUDA, 컴파일러, GPU
학번 : 2018-24786

목차

그림 목차

표 목차

제 1장 서론

이종 시스템(heterogeneous system)은 CPU뿐만 아니라 GPU, FPGA, DSP 등 다양한 하드웨어가 활용되는 컴퓨팅 시스템으로써 저전력, 고성능을 특징으로 한다. 오늘날 대부분의 고성능 클러스터는 이종 시스템을 바탕으로 한 병렬 컴퓨팅을 기초로 한다. 나아가 과학 계산, 기계학습 등 대규모 데이터와 병렬 연산을 요구하는 응용이 중요해지면서, 이종 시스템을 효율적으로 사용하기 위한 프로그래밍 모델이 요구되고 있다.

그러나 이종 시스템을 위한 프로그램을 작성하는 일은 많은 어려움이 따른다. 다양한 하드웨어를 사용하기 위해 다양한 프로그래밍 모델과 언어를 학습해야 한다. 뿐만 아니라 시스템의 구성요소가 변화할 경우 목표하는 성능을 달성하기 위해 새롭게 프로그램을 작성해야 하는 경우가 빈번하다.

OpenMP[1]는 공유 메모리 병렬 처리 프로그래밍 모델로서 오래 전부터 CPU를 활용한 고성능 컴퓨팅 분야에서 사실상 표준으로 자리 잡고 있다. OpenMP는 하나의 마스터 스레드가 여러 스레드를 생성하여 작업을 처리한 뒤 동기화하는 fork-join 모델을 채택하고 있다. fork-join 모델은 Pthread 등 시스템 콜 수준의 스레드 라이브러리보다 쉽고 직관적이라는 장점이 있다. 나아가 OpenMP는 한번 작성한 프로그램이 여러 시스템과 하드웨어를 아울러 높은 성능을 보이도록 하는 성능 이식성을 핵심 목표로 하며, 이를 위해 컴파일러 기술을 활용한다.

OpenMP 4.0은 fork-join 모델을 확장하여 이종 시스템에 대한 지원이 대폭 강화하였으며, 특히 device constructs를 이용해 GPU, co-processor(예. 인텔 Xeon-phi) 등 여러 가속기를 활용할 수 있게 되었다. 본 논문은 OpenMP 4.5의 device construct를 사용해 작성된 C 프로그램을 CUDA[2] 코드로 변환하는 컴파일러와 변환된 프로그

램을 위한 런타임 라이브러리를 제시한다. 그림 1은 본 논문이 제시한 프레임워크의 작동 과정을 보여준다. 우선, 컴파일러는 OpenMP device construct를 사용해 작성된 C 프로그램을 입력으로 받아 동일한 시맨틱의 CUDA 프로그램을 생성한다. 메모리 할당 및 복사, 커널 실행 등 런타임에 실행되는 기능들은 프레임워크에 포함된 런타임 라이브러리를 통해 제공된다. 생성된 CUDA 프로그램은 상용 CUDA 컴파일러 (nvcc, clang)를 백엔드로 사용해 바이너리로 컴파일 된다.

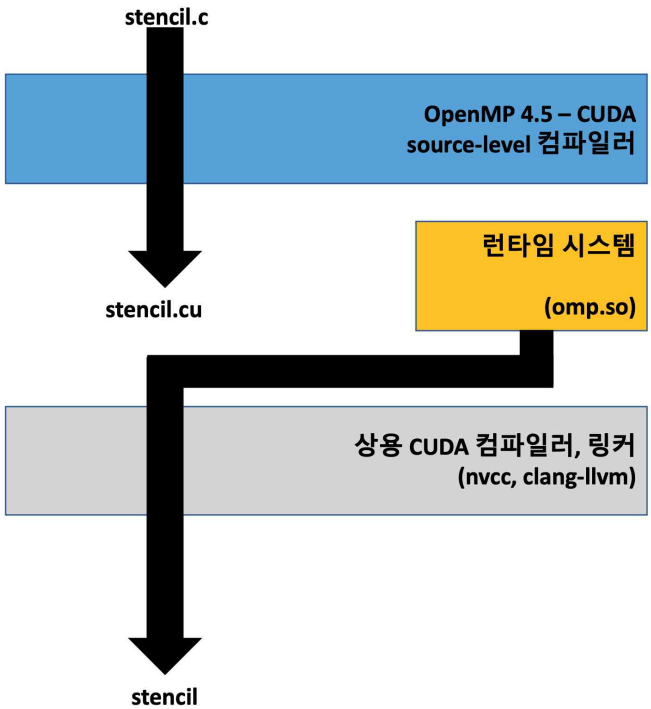


그림 1 본 논문이 제시하는 프레임워크의 작동 과정

본 논문의 학술적 기여는 다음과 같다.

- OpenMP 4.5 프로그램을 CUDA 프로그램으로 변환하는 source-level 컴파일러를 제시한다. source-level 변환을 통해 사용

자는 실제 생성된 CUDA 프로그램을 확인할 수 있으며, 따라서 컴파일러가 못한 최적화 등의 수작업을 추가적으로 진행할 수 있다. 나아가 향후 source-level 시맨틱을 활용한 컴파일러 최적화 기술을 개발하는 토대가 된다.

- OpenMP 4.5 device constructs를 지원하는 런타임 시스템과 다양한 런타임 최적화 기술들을 제안한다. 제시된 런타임 시스템 디자인과 최적화 기술들은 NVIDIA GPU 뿐만 아니라 다른 하드웨어에도 쉽게 적용할 수 있을 것으로 예상된다.
- 표준 speaccel 1.2 벤치마크를 사용한 실험 결과를 제시하고 순차 코드 및 gcc7과의 자세한 비교 분석을 제시하여 OpenMP device constructs의 성능 특성을 드러낸다. 나아가 OpenMP 4.5 버전이 지니고 있는 한계를 지적한다.

본 논문은 본 장(제 1장 서론)을 포함해 총 8개의 장으로 구성되어 있으며, 각 장의 구성은 다음과 같다.

제 2장에서는 연구 배경과 관련 연구를 살펴본다. 특히 이종 시스템을 위한 프로그래밍 모델과 GPU를 위한 OpenMP 프레임워크들을 조망한다.

제 3장과 제 4장은 본 논문의 바탕을 이루는 OpenMP 및 CUDA 프로그래밍 모델에 대해 설명한다. 제 3장은 NVIDIA GPU를 위한 CUDA 프로그래밍 모델을 기술한다. CUDA의 실행 모델 및 메밀 모델, 동기화 모델을 소개한다.

제 4장에서는 OpenMP 프로그래밍 모델을 살펴본다. 특히 device constructs를 통해 확장된 fork-join 실행 모델과 메모리 모델 및 동기화 모델에 주목한다.

제 5장과 제 6장은 본 논문이 제시한 프레임워크에 대해 자세하게 설명한다. 제 5장에서는 OpenMP device constructs를 CUDA 프로그램으로 변환하는 source-level 컴파일러를 설명한다. 특히 loop를 병렬화 하는 알고리즘, 메모리 시맨틱을 지키기 위한 메모리 관

리 기법, 그리고 동기화와 스레드 간 통신 기법을 제시한다.

제 6장에서는 프로그램 실행을 위한 런타임 시스템과 런타임 최적화를 위한 기법들을 살펴본다. 먼저 커널 실행 시간을 단축시키기 위해 사용된 동적 구간 트리를 이용한 커널 스케쥴링 기술과 동적 커널 선택을 이용한 스레드 스케쥴링 기술을 살펴본다. 나아가 메모리 관리 및 통신 시간을 단축하기 위한 Buddy 할당자와 UDTE 알고리즘을 살펴본다.

제 7장에서는 실험 환경을 제시하고, spec-accel 1.2 벤치마크를 이용한 실험 결과를 분석한다. 비교 대상은 순차 코드와 gcc7 컴파일러와 libgomp 런타임 시스템을 사용한 프로그램이며, 실험 결과 본 논문이 제시한 프레임워크는 평균적으로 순차 코드 대비 53.4배, gcc7 대비 6.75배 빠른 성능을 보였다. 나아가 제 7장에서는 OpenMP device constructs의 문제와 한계도 분석한다.

마지막으로 제 8장에서는 최종 결론과 논의 및 남겨진 과제들을 지적하며 마무리한다.

제 2장 배경 및 관련 연구

OpenMP[1]는 공유 메모리 병렬 처리 프로그래밍 모델이며, 고성능 컴퓨팅 분야에서 사실상 표준이다. OpenMP 버전 4.0에서 도입된 device constructs는 이종 시스템에 대한 지원을 강화하기 위한 조처였으며, GPU, co-processor 등 여러 가속기를 활용해 성능 최적화를 꾀할 수 있도록 한다. OpenMP는 작성한 프로그램이 여러 시스템과 하드웨어를 걸쳐 높은 성능을 보이도록 하는 성능 이식성(performance portability)을 목표로 하고 있으며, 이를 위해 효율적인 컴파일러 및 런타임 기술이 절실한 상태이다.

본 연구는 OpenMP 4.5의 device constructs를 이용해 작성된 C 프로그램을 CUDA[2] 코드로 번환하는 컴파일러와 프로그램의 실행에 필요한 런타임 시스템을 제안하고자 한다.

OpenMP 프로그램의 GPU 실행과 관련한 대표적 연구로는 [3, 4]가 있다.

[3]은 OpenMP 3.0 프로그램을 CUDA 소스로 변환하는 컴파일러 기술을 제시한다. 루프 형태로 주어진 OpenMP 프로그램을 CUDA 커널로 변환하며, 이 과정에서 메모리 레이아웃 변환, 메모리 coalescing 등 다양한 최적화 기술을 적용한다. 본 논문은 OpenMP 4.0부터 도입된 device constructs를 이용해 작성된 프로그램을 목표로 하고 있으며, 컴파일러 최적화보다 런타임 최적화에 집중한다는 점에서 차이가 있다.

[4]는 OpenMP 4.0 프로그램을 OpenCL 프로그램으로 번역하여 GPU 상에서 실행한다. 나아가 페이지 보호 기능을 이용해 불필요한 메모리 복사를 없애 성능 향상을 보여준다. 본 논문은 OpenMP 4.5 프로그램을 대상으로 하며 주어진 프로그램을 OpenCL이 아닌 CUDA 프로그램으로 변환한다는 점에서 차이가 있다.

상용 컴파일러 중 OpenMP device contruct를 지원하는 것으로는 gcc version 7[4]이 있다. gcc는 본 연구에서 제시하는 소스 레벨 번역

기와 달리 OpenMP 프로그램을 번역한 결과를 소스코드로 산출하지 않는다는 차이가 있다. gcc7과 본 연구의 프레임워크 간 성능 차이는 제 7장을 참조하라.

제 3장 CUDA 프로그래밍 모델

본 장은 NVIDIA GPU를 위한 CUDA 프로그래밍 모델을 살펴본다. CUDA의 실행 모델 및 메밀 모델, 동기화 모델을 소개한다. 자세한 사항은 [2]를 참조하라.

3.1 실행 모델

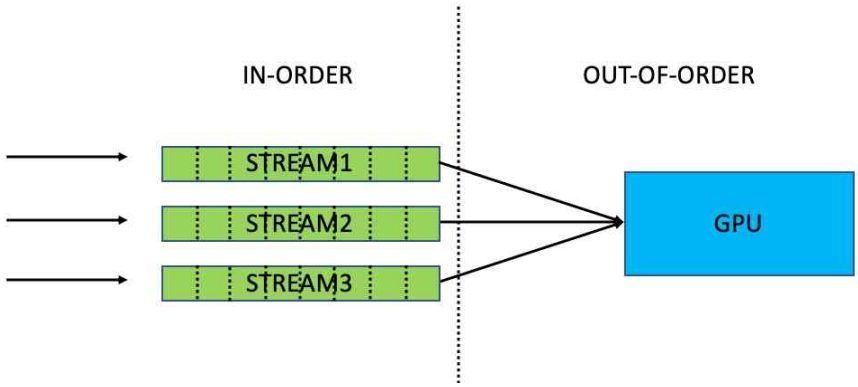


그림 2 CUDA 실행 모델

CUDA는 NVIDIA GPU를 위한 가속기 프로그래밍 모델이며, 직관적인 실행 모델을 지니고 있다. 사용자는 GPU상에서 실행되는 커널(kernel)을 작성한다. 작성된 커널은 CUDA 스트림(stream)에 삽입하는 것으로 비동기적으로 실행할 수 있다. 나아가 커널뿐만 아니라 메모리 복사 연산 등 통신 연산도 스트림에 연산을 삽입하여 이루어진다. 사용자는 API 호출을 통해 스트림에 삽입된 모든 연산이 종료하기를 기다릴 수 있다([2], cudaStreamSynchronize). 나아가 각 CUDA 이벤트를 활용해 연산 단위의 동기화도 수행할 수 있다([2], cudaEventSynchronize).

하나의 CUDA 스트림은 in-order queue이며, 따라서 스트림에 삽입된 순서대로 실행된다. 삽입된 커널은 동일한 스트림에 먼저 삽입된 커널이 끝난 뒤에 실행된다는 것이 보장된다. 서로 다른 CUDA

스트림은 독립적이며, 여러 스트림의 queue에 커널이 들어있다면, 사용자는 다음 순서에 실행될 커널이 무엇인지 정확히 알 수 없다. 일반적으로 의존성이 있는 커널들은 같은 스트림에 의존성이 없는 커널들은 서로 다른 스트림에 삽입한다. 이를 통해 불필요한 동기화를 없애고 연산-통신 overlapping 등 효율적인 성능을 달성한다.

CUDA 커널 실행하기 위해서는 사용할 스레드 블록(thread block)과 스레드(thread)의 개수를 설정해야 한다. 스레드는 커널을 실행하는 기본 단위이며 각 스레드는 독립적으로 동작한다. 스레드 블록이란 동기화할 수 있는 스레드들의 집합이다. 스레드 블록들의 묶음을 그리드(grid)라고도 한다.

3.2 메모리 모델 및 동기화

기본적으로 CUDA는 이산적(discrete) 메모리 모델을 지니고 있으며, CPU 메모리 주소 공간과 GPU 메모리 주소 공간은 분리된다.[1]이 때, CPU 메모리를 호스트 메모리, GPU 메모리를 디바이스 메모리라 한다. 호스트 메모리와 디바이스 메모리 간의 통신은 API 호출을 통해 스트림을 이용해 메시지 전달의 형태로 이루어진다. CUDA는 relaxed 메모리 컨시스턴시 모델을 채택하고 있으며, 메모리 배리어 연산을 제공한다.

동일한 스레드 블록에 속하는 스레드들은 API 호출을 통해 명시적으로 동기화할 수 있지만([2], __syncthreads), 서로 다른 스레드 블록에 속하는 스레드들 간에는 명시적으로 동기화할 수 없다 (그림 3). 단, 커널 실행의 종료를 통해 암묵적으로 모든 스레드가 실행을 종료했다는 것을 알 수 있다.

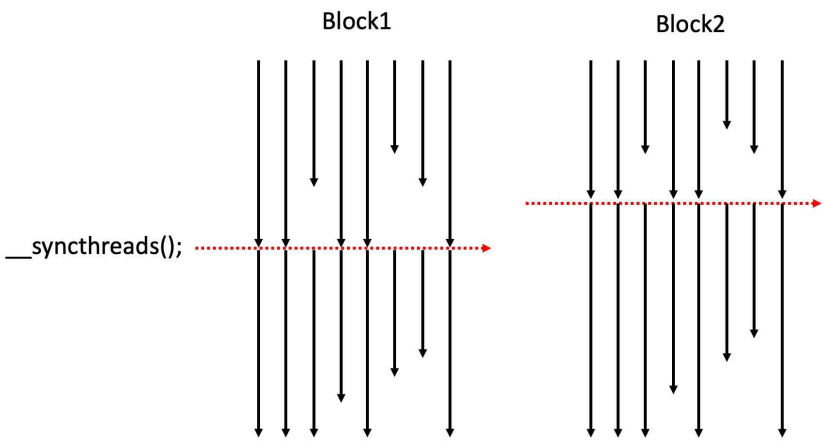


그림 3 CUDA 동기화 모델

1) 특별한 경우는 [2]의 Unified memory.항목을 참고하라.

제 4장 OpenMP와 Device Constructs 모델

본 장에서는 OpenMP 프로그래밍 모델을 살펴본다. 먼저, OpenMP의 중요한 특징인 fork-join 모델을 살펴본다. 나아가 OpenMP 4.0 추가된 device constructs와 확장된 fork-join 실행 모델과 메모리 모델 및 동기화 모델을 살펴본다.

4.1 실행 모델

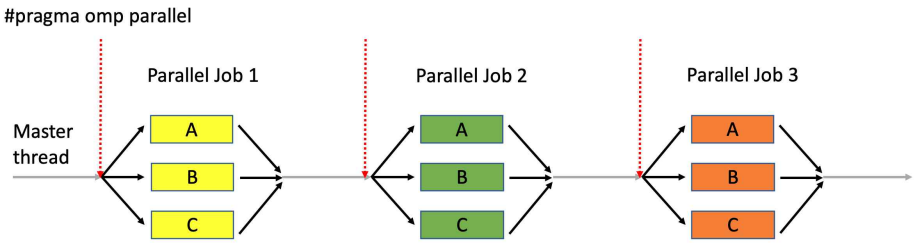


그림 4 OpenMP fork-join 모델

OpenMP[1]는 공유 메모리 병렬 처리 프로그래밍 모델로서, 순차 코드로 작성된 프로그램을 컴파일러 프라그마(pragma, or constructs)를 사용해 병렬화 한다. 기본적으로 OpenMP는 멀티코어로 구성된 공유 메모리 시스템을 가정하고 있으며, 병렬 처리는 각 코어에서 작동하는 스레드(thread)를 통해 수행된다. 사용자는 순차 코드로 작성된 프로그램 중 병렬처리하려는 영역에 적절한 parallel 프라그마를 삽입한다. OpenMP 컴파일러는 표준에 정해진 프라그마의 시맨틱에 따라 스레드 생성, 스케쥴링 및 동기화하는 코드를 생성한다. 컴파일된 OpenMP 프로그램을 실행하면 하나의 마스터 스레드가 생성되고, 마스터 스레드는 메인 함수를 호출한다. 사용자가 프라그마를 지정하지 않은 영역은 일반적인 순차 프로그램과 동일하게 실행된다. 사용자가 프라그마를 지정한 영역을 만나면, 진입하

기 직전에 스레드를 생성하고(fork) 실행이 끝난 뒤 생성되었던 스레드들은 동기화하고 소멸한다(join). 이러한 실행 모델을 fork-join 모델이라 한다 (그림 2 참조).

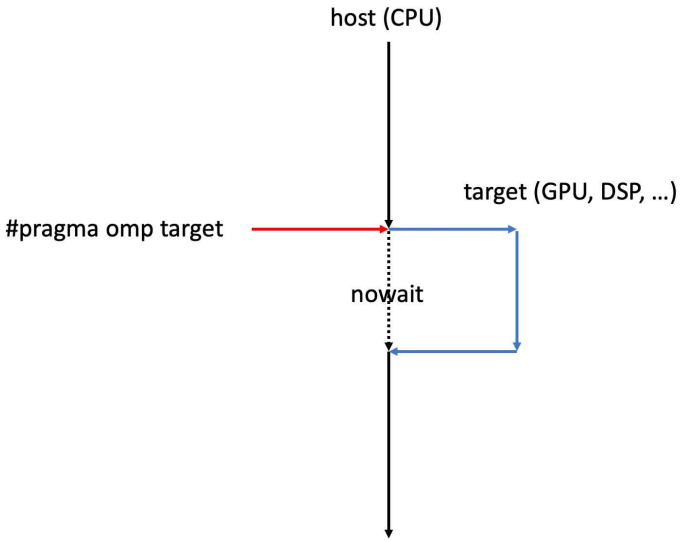


그림 5 OpenMP 타겟 오프로딩 실행 모델

OpenMP 4.0은 이종 시스템에 대한 지원을 강화하였다. 먼저, OpenMP 4.0은 device constructs를 추가하여 호스트 디바이스(CPU)가 아닌 타겟 디바이스(GPU, DSP, 등)를 이용한 병렬화를 지원하는 타겟 오프로딩(target offloading) 기능을 추가하였다. 타겟 오프로딩이 지정된 영역은 CPU를 포함한 임의의 연산 장치에 스케쥴링 된다. 기본적으로 호스트 스레드는 타겟 오프로딩 영역 실행이 끝날 때까지 블록킹되지만, 명시적으로 비동기 프라그마를 지정할 경우 호스트 영역과 타겟 오프로딩 영역이 비동기적으로 실행될 수 있다 (그림 3 참조).

한편, OpenMP 4.0은 기존의 fork-join 모델을 확장하여 타겟 오프로딩을 위한 새로운 fork-join 모델을 제시하였다. 일반적으로

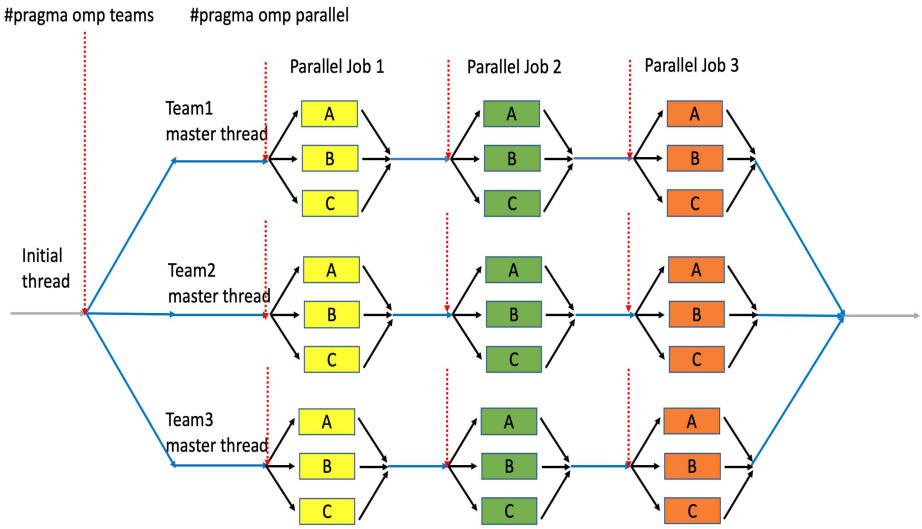


그림 6 확장된 OpenMP fork-join 모델

GPU 등 가속기는 많은 양의 연산 코어를 지니고 있으며, 성능을 위해 코어 간 동기화 등에 제약이 있다. 예컨대, CUDA의 경우 스레드 블록간의 동기화는 커널 실행 종료 시에만 이루어진다(제 4장 참조). 기존의 fork-join 모델은 잦은 스레드간 동기화를 유발하며, 이는 성능에 큰 문제를 가져왔다.

OpenMP 4.0은 팀(team)이라는 새로운 스케쥴링 단위를 도입하여 문제를 해결하였다. 모든 스레드는 하나의 팀에 속하며, 같은 팀에 속하는 스레드 간에는 동기화가 가능하지만, 다른 팀에 속하는 스레드 간에는 동기화할 수 없다. 사용자는 device constructs를 통해 여러 개의 팀들을 생성할 수 있으며, 이는 기존에 스레드를 생성하는 과정과 유사하다.

OpenMP 프로그램 실행 시 이니셜 스레드(initial thread)와 이니셜 팀(initial team)이 생성되며, 이니셜 스레드는 메인 함수를 호출한다. 기존 fork-join 모델과 마찬가지로 순차 영역은 순차 프로그램과 동일하게 동작한다. 팀 생성 프라그마가 지정된 영역에 진입할 경우, 이니셜 스레드는 수 개의 팀과 각 팀의 마스터 스레드를 생성

하며, 이 때 각 팀들은 독립적으로 스케쥴링된다. 각 팀의 마스터 스레드가 스레드 생성 프라그마가 지정된 영역에 진입할 경우 마스터 스레드와 동일한 팀에 속하는 수 개의 스레드를 생성한다. 기존의 fork-join 모델과 유사하게 팀 생성 프라그마가 지정된 영역에서 탈출할 경우에 생성된 팀들은 동기화하고 소멸한다 (그림 4 참조).

4.2 메모리 모델 및 동기화

OpenMP는 공유 메모리 모델을 가정하며, 따라서 사용자는 여러 스레드간의 race condition 등 동기화 관련 문제를 방지하기 위해 락킹 메커니즘을 사용해야 한다. OpenMP는 relaxed 컨시스턴시 모델을 채택하고 있으며, 메모리 배리어 연산을 제공한다.

한편, 타겟 오프로딩을 사용할 경우에도 여전히 공유 메모리 모델이 적용된다. 동일한 디바이스안의 통신은 공유 메모리 모델이 그대로 적용된다. 그러나 서로 다른 디바이스는 독립된 주소 공간을 가지고 있으므로, 사용자는 명시적으로 각 디바이스가 사용할 주소 공간을 맵핑해주어야 한다.

동일한 팀에 속하는 스레드들은 프라그마를 통해 명시적으로 동기화할 수 있지만([1], #pragma omp barrier), 서로 다른 팀에 속하는 스레드들 간에는 명시적으로 동기화할 수 없다 (그림 3 참조). 단, 타겟 오프로딩 실행의 종료를 통해 암묵적으로 모든 스레드가 작업을 완료했다는 것을 보장받을 수 있다.

제 5장 Source-level 변환 컴파일러

본 장에서는 OpenMP device constructs를 CUDA 프로그램으로 변환하는 source-level 컴파일러를 설명한다.

일반적으로 OpenMP는 많은 연산이 요구되는 순차 루프(loop)를 병렬화하기 위해 사용된다. 5.1 절에서는 device constructs를 사용했을 경우 순차 루프가 어떻게 변환되는지를 살펴본다.

한편, OpenMP 프라그마는 private, shared 등 다양한 메모리 시맨틱을 지원한다. 5.2 절에서는 메모리 시맨틱을 지키기 위한 메모리 맵핑 기법을 살펴본다.

5.3절은 확장된 fork-join 모델에서 스레드 간 동기화 및 통신을 지원하기 위한 변환 과정을 살펴본다.

5.1 루프 변환

그림 7은 OpenMP device construct를 사용해 코드를 병렬화한 모습을 나타낸다. 대부분의 for 루프는 *teams distribiute parallel for* 프라그마를 사용해 병렬화 한다. *teams distribute*는 수 개의 team들을 생성하여 for 루프를 병렬화하며, *parallel for* 프라그마는 각 팀에서 여러 개의 스레드들을 생성하여 할당된 for 루프를 병렬화한다. 이때 동일한 팀에 속하는 스레드들은 서로 동기화할 수 있으며, 서로 다른 팀에 속하는 스레드들은 그렇지 않다 [1].

그림 7 (c)는 saxpy 프로그램의 CUDA 버전을 보여준다. CUDA 프로그래밍 모델에 따르면, 커널들은 스레드와 스레드 블록을 생성하여 실행된다. 하나의 스레드 블록은 여러 개의 스레드 구성되어 있으며, 실행된 여러 개의 블록 집합을 그리드라 칭한다. 동일한 블록에 속하는 스레드들은 동기화할 수 있으며, 서로 다른 블록 에 속하는 스레드들은 동기화할 수 없다.

이러한 OpenMP 팀과 CUDA 스레드 블록 간의 대응에 기초하면 그림 7 (b)는 자연스럽게 그림 7 (d)와 같이 변환된다. 호스트 코드의 OpenMP 타겟 오프로딩 영역은 메모리 할당, 복사 및 커널 실행 함수들로 교체되며, 해당 영역을 변환한 커널 함수가 새롭게 생성된다. nowait clause가 명시적으로 설정되지 않은 타겟 오프로딩 영역은 커널 실행 혹은 계산 결과를 복사해오는 과정에서 동기화된다.

이 때, get_dev_ptr, copy_to_device, copy_from_device, release_ptr 등 함수는 런타임 라이브러리 함수들로, 할당된 디바이스 메모리를 읽어오거나 반환하며, 호스트-디바이스 간의 메모리 복사 연산을 수행하다. 이들은 내부적으로 CUDA API와 기타 자료 구조를 이용해 구현되어 있다.

한편, 그림 7 (d)와 같이 런타임 라이브러리 함수들은 명시적으로 CUDA 스트림을 관리하지 않는다. CUDA 스트림은 제 6장에서 볼 수 있듯이, 효율적인 알고리즘을 통해 런타임 시스템이 내부적으로 관리한다.

```
1  void saxpy(int n, float a, const float *x, float *y) {
2      for (int i =0; i < n; i ++) {
3          y[i] = a * x[i] + y[i];
4      }
5  }
```

(a) serial saxpy

```
1  void saxpy(int n, float a, const float *x, float *y) {
2      #pragma omp target                    \
3      teams distribute parallel for         \
4      map(to:x[0:n]) map(tofrom: y[0:n])
5      for (int i =0; i < n; i ++) {
6          y[i] = a * x[i] + y[i];
7      }
8  }
```

(b) OpenMP saxpy

```
1  __global__ void
2  saxpy_kernel(int n, float a, const float *x, float *y) {
3      int i = blockIdx.x * blockDim.x + threadIdx.x;
4      if (i >= n) return; // boundary check
5      y[i] = a * x[i] + y[i];
6  }
```

(c) CUDA saxpy

```
1  void saxpy(int n, float a, const float *x, float *y) {
2      float *x_dev = get_dev_ptr(sizeof(float) * n);
3      float *y_dev = get_dev_ptr(sizeof(float) * n);
4      copy_to_device(x_dev, x, sizeof(float)*n);
5      copy_to_device(y_dev, y, sizeof(float)*n);
6      launch_kernel(saxpy_kernel, div_up(n, 128), 128,
7                    n, a, x_dev, y_dev);
8      copy_from_device(y, y_dev, sizeof(float)*n);
9      release_ptr(x_dev);
10     release_ptr(y_dev);
11 }
```

(d) 변환된 saxpy

그림 7 serial, OpenMP, CUDA saxpy 코드 예제

5.2 메모리 관리

OpenMP는 공유 메모리 모델에 기반을 두고 있지만, 쉽고 효율적인 프로그래밍을 위해 다양한 메모리 시맨틱을 지원한다. 대표적으로 (팀을 포함하여) 스레드를 생성하는 프라그마는 private 혹은 shared clause을 지니고 있다. private으로 선언된 변수는 thread-local 변수로 지정되며, 새롭게 생성된 각 스레드가 독립적으로 저장공간을 할당받는다. 반면 shared로 선언된 변수는 새롭게 생성된 스레드들간에 공유된다.

CUDA도 유사하게 thread-local 변수와 shared 변수를 지닐 수 있다. 선언 시 __shared__ qualifier가 설정된 변수는 동일한 스레드 블록에 속하는 스레드들간에 공유되며, 설정되지 않은 변수는 각 스레드가 독립적으로 저장 공간을 할당받는다. 결론적으로 private 변수의 경우는 CUDA 커널 안에서 변수를 선언하는 것과 대응되고, 팀 내부의 shared 변수의 경우는 CUDA shared 변수에 대응한다.

그러나 OpenMP 표준[1]은 relaxed 메모리 컨시스턴시 하에서 팀 간의 shared 변수도 허용한다. relaxed 메모리 컨시스턴시 모델 때문에 팀 간의 shared 변수가 포함된 영역은 보조 디바이스 메모리(auxiliary device memory)를 할당해 처리할 수 있다.

한편, reduction 변수들도 shared 변수와 유사하게 관리된다. 특히, 팀 간의 reduction 변수도 보조 디바이스 메모리를 요구한다. 컴파일러는 팀 간의 reduction 변수를 만나면 각 팀이 사용할 수 있는 디바이스 메모리를 할당하는 코드를 생성한다. 생성된 커널은 이러한 디바이스 메모리에 연산을 수행한다. 또한, 커널 실행이 끝난 뒤 동기화 지점에 할당된 보조 디바이스 메모리를 호스트로 읽어 들이는 코드를 생성한다. 마지막으로 읽어 들인 값에 대한 reduction 연산을 수행하는 CPU 코드를 생성한다.

5.3 동기화(Synchronization)

확장된 fork-join 모델에 따르면, 동일한 팀에 속하는 스레드들 간에는 동기화할 수 있지만, 서로 다른 팀에 속하는 스레드들 간에는 명시적으로 동기화할 수 없다 (제 4장 참조). 사실, 팀은 CUDA 스레드 블록에 대응하고, 스레드는 CUDA 스레드에 대응한다는 것을 쉽게 확인할 수 있다. 결국 fork-join 모델로부터 비롯되는 묵시적 동기화는 CUDA 커널 실행 모델에 의해 자동적으로 만족된다.

한편, OpenMP 표준[1]에 따르면 명시적 동기화(barrier)는 어떤 한 팀에 속하는 모든 스레드가 실행하거나, 그 팀에 속하는 어스레드도 실행해서는 안된다. 유사하게, CUDA 명세[2]에 따르면, 명시적 동기화(__syncthreads)는 어떤 한 스레드 블록에 속하는 모든 스레드가 실행하거나, 그 블록에 속하는 어떠한 스레드도 실행해서는 안된다. 결론적으로 팀과 스레드 블록의 대응으로 인해 OpenMP와 CUDA의 명시적 동기화는 쉽게 달성된다 (그림 8 참조).

```
1#pragma omp target teams distribute parallel for
2for (int i =0; i < n; i ++) {
3       ... /* body 1 */
4   #pragma omp target
5   for (int j =0; j < n; j ++) {
6       .... /* body 2 */
7   }
8 }
```

```
1int i = blockIdx.x * blockDim.x + threadIdx.x;
2if (i >= n) return; // boundary check
3... /* body 1 */
4__syncthreads();
5for (int j =0; j < n; j ++) {
6    .... /* body 2 */
7}
8}
```

그림 8 OpenMP-CUDA 동기화 예제

제 6장 런타임 시스템 최적화

본 장에서는 프로그램 실행을 위한 런타임 시스템과 런타임 최적화를 위한 기법들을 살펴본다.

먼저 커널 실행 시간을 단축시키기 위해 사용된 동적 구간 트리를 이용한 커널 스케쥴링 기술과 동적 커널 선택을 이용한 스레드 스케쥴링 기술을 살펴본다. 나아가 메모리 관리 및 통신 시간을 단축하기 위한 Buddy 할당자와 UDTE 알고리즘을 살펴본다.

6.1 커널 실행 최적화

6.1.1 동적 구간 트리(Dynamic Interval Tree)

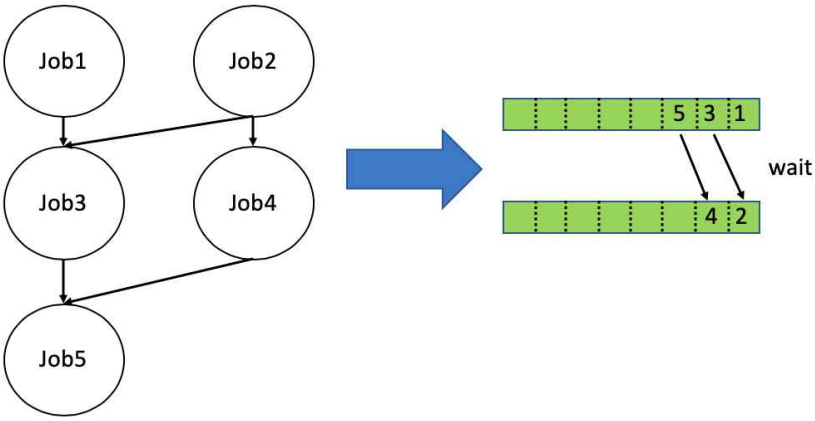
제 5장에서 살펴본 바와 같이 OpenMP 타겟 오프로딩은 CUDA 메모리 복사 및 커널 실행과 대응한다. 이 때, 하나의 CUDA 스트림만 사용할 경우 in-order 특성으로 인해 불필요한 동기화 시간이 소모된다. 불필요한 동기화를 줄이기 위해서는 1) 커널 (혹은 메모리 복사) 작업 간의 의존성을 파악하고, 2) 의존성이 없는 작업들은 서로 다른 스트림에 삽입되도록 해야 한다. 그러나 OpenMP device constructs는 동적 메모리 맵핑을 허용하므로 컴파일러 분석만을 통해서는 작업 간의 의존성을 완벽히 파악할 수 없다.

본 논문에서 제시하는 런타임 시스템은 동적 인터벌 트리(dynamic interval tree)를 이용해 런타임에 작업 간의 의존성을 파악한다. 동적 구간 트리는 구간 트리로 현재 타겟 디바이스에 의해 사용되고 있는 메모리 영역에 대한 정보를 관리한다. 각 노드는 메모리 영역의 집합과 해당 영역을 사용하고 있는 작업들의 집합을 저장하고 있다. 부모 노드의 메모리 영역 집합과 작업 집합은 각각 자식 노드들의 메모리 영역 집합들의 합집합과 작업 집합들의 합집합

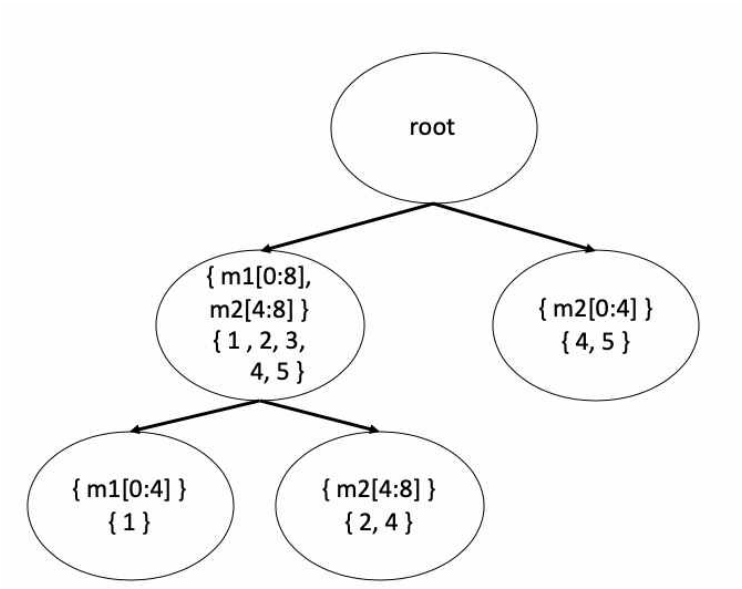
이다. 루트 노드는 전체 메모리 영역과 모든 작업을 의미한다.

런타임 시스템은 메모리 작업을 수행하기 전에 동적 구간 트리에 메모리 영역을 키로 검색을 수행한다. 만약, 해당 키를 원소로 가지고 있는 노드가 있으면 해당 영역을 사용 중인 작업 중 하나를 포함하는 스트림에 메모리 작업을 삽입하고, 해당 스트림에 포함되지 않는 작업들은 CUDA 이벤트를 통해 기다린다. 그렇지 않으면, 해당 정보를 트리에 업데이트하고 비어있는 스트림에 작업을 실행한다.

그림 9의 (a)의 왼쪽과 같이 작업들이 의존성을 지닌다고 하자. 이 때, Job1과 Job2는 서로 다른 스트림을 이용해 병행적으로 (concurrently) 실행될 수 있다. Job3는 Job1과 Job2의 완료를 기다려야 하고, Job4는 Job2의 완료를 기다려야 하며, Job5는 Job3와 Job4의 완료를 기다려야 한다. 이 경우 가능한 최적의 스케쥴링은 그림 9 (a)의 오른쪽과 같으며, 최종 동적 구간 트리는 그림 9 (b)와 같다.



(a) 작업 의존성과 스트림 스케쥴



(b) 동적 구간 트리 상태

그림 9 동적 구간 트리 예제

6.1.2 동적 커널 선택(Dynamic Kernel Selection)

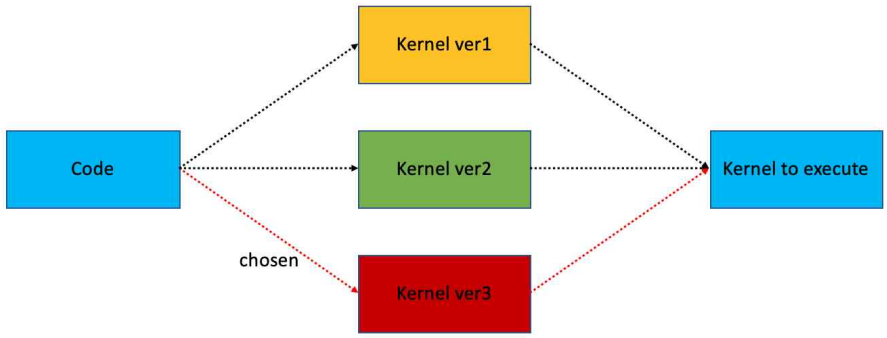


그림 10 동적 커널 선택의 도식

OpenMP device constructs는 팀, 스레드, SIMD 등 다양한 시맨틱을 제공하여 프로그래밍을 쉽게 하고 다양한 하드웨어를 지원하지만, 반대로 구현의 자유도가 높아 성능 이식성 및 성능 안정성이 낮아지는 문제가 발생한다.

그림 11, 그림 12, 그림 13는 동일한 연산을 수행하는 타겟 오프로딩 코드를 다양하게 구현한 모습을 나타낸다. 외부 루프와 내부 루프의 사이즈는 런타임에 결정되기 때문에, 사용자가 다양한 하드웨어가 최적의 코드를 선택하는 것은 쉽지 않다.

뿐만 아니라 컴파일러가 최적의 스케쥴링을 만들어내는 것도 쉽지 않다. 예를 들어 그림 10에서 외부 루프의 사이즈가 내부 루프보다 훨씬 클 경우 모든 스레드를 외부 루프에 할당하고 내부의 SIMD 프라그마는 무시할 수 있다. 반대로 내부 루프의 사이즈가 훨씬 클 경우 성능을 최대한 높이기 위해 내부 루프에 워프를 할당하고, 외부 루프에 워프 마스터를 할당할 수 있다. 이렇듯, 성능 이식성을 높이는 코드를 작성하는 것과 작성된 코드의 성능을 최대한

끌어내는 것은 어렵다.

이러한 문제를 해결하기 위해 프레임워크는 컴파일 단계에서 여러 버전의 커널을 생성해놓는다. 실제 실행할 때는 파라미터 값을 읽어 병렬성의 정도가 높은 커널을 선택해 실행하도록 한다(그림 10 참조).

```
1 end = lastrow - firstrow + 1;
2 #pragma omp target  \
3 map(rowstr[:0], colidx[:0], a[:0], p[:0], q[:0])
4 #pragma omp teams distribute parallel for \
5     private(tmp1, tmp2, sum)
6 for (j = 0; j < end; j++) {
7     tmp1 = rowstr[j];
8     tmp2 = rowstr[j+1];
9     sum = 0.0;
10    #pragma omp simd reduction(+:sum) private(tmp3)
11    for (k = tmp1; k < tmp2; k++) {
12        tmp3 = colidx[k];
13        sum = sum + a[k] * p[tmp3];
14    }
15    q[j] = sum;
16 }
```

그림 11 중첩 루프 OpenMP 구현 예제 1

```
end = lastrow - firstrow + 1;
#pragma omp target  \
map(rowstr[:0], colidx[:0], a[:0], p[:0], q[:0])
#pragma omp teams distribute
for (j = 0; j < end; j++) {
    tmp1 = rowstr[j];
    tmp2 = rowstr[j+1];
    sum = 0.0;
    #pragma omp parallel for simd  \
    reduction(+:sum) private(tmp3) shared(tmp1, tmp2)
    for (k = tmp1; k < tmp2; k++) {
        tmp3 = colidx[k];
        sum = sum + a[k] * p[tmp3];
    }
    q[j] = sum;
}
```

그림 12 중첩 루프 OpenMP 구현 예제 2

```
end = lastrow - firstrow + 1;
#pragma omp target data  \
map(rowstr[:0], colidx[:0], a[:0], p[:0], q[:0])
for (j = 0; j < end; j++) {
    tmp1 = rowstr[j];
    tmp2 = rowstr[j+1];
    sum = 0.0;
    #pragma omp target teams parallel for simd  \
    reduction(+:sum) private(tmp3) shared(tmp1, tmp2)
    for (k = tmp1; k < tmp2; k++) {
        tmp3 = colidx[k];
        sum = sum + a[k] * p[tmp3];
    }
    q[j] = sum;
}
```

그림 13 중첩 루프 OpenMP 구현 예제 3

6.2 메모리 관리 최적화

6.2.1 버디 할당자

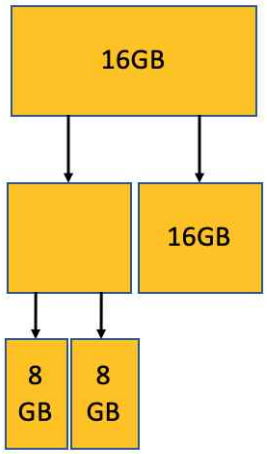


그림 14 버디 할당자

CUDA 디바이스 메모리를 할당할 때 시스템 콜 호출 및 메모리 맵핑 관리 등에 많은 시간이 소요된다. 때문에 디바이스 메모리 할당 함수를 빈번하게 호출하는 것은 많은 비용을 수반한다. 특히, 제5장에서 살펴본 대로 디바이스 메모리 관리는 많은 수의 보조 메모리(auxiliary memory)를 요구하며, 따라서 효율적인 디바이스 메모리 할당자가 필요하다.

본 논문의 프레임워크는 효율적인 메모리 관리를 위해 운영체제에서 사용되는 버디 할당자(buddy allocator)를 사용한다[7]. 런타임 시스템은 초기화 과정에서 사용가능한 모든 디바이스 메모리를 할당한다(그림 11, 16GB). 실행 중 요청되는 크기에 따라 메모리를 트리 구조로 분할하여 할당한다. 이후 메모리가 반환될 경우 자매(sibling) 노드를 본 뒤 합치기(coalescing) 작업을 수행한다.

6.2.2 UDTE

OpenMP의 fork-join 모델은 불필요한 메모리 통신을 수반하기도 한다. 표 9에서와 같이 동일한 디바이스에 오프로딩 되는 두 루프가 사이에 외부 함수가 존재하고, 두 루프가 외부 전역 변수를 사용한다고 하자. 외부 함수의 구현을 모르는 사용자는 각 루프를 동기적으로 동작하도록 해야 하며, 따라서 많은 양의 메모리 통신이 발생한다. 이 때, 컴파일러 분석은 alising 및 link의 문제로 인해 한계를 지닌다.

```
1 extern int *buf; /* external global variable */
2 #pragma omp target teams parallel for simd \
3     map(buf[0:n])
4     for (i = 0; i < n; i++) {
5         buf[i] += i;
6     }
7 foo(); /* external function */
8  #pragma omp target teams parallel for simd \
9     map(buf[0:n])
10    for (i = 0; i < n; i++) {
11        buf[i] += i;
12    }
```

그림 15 UDTE를 요구하는 코드 예제

[4]은 이러한 문제를 해결하기 위해 Unnecessary Data Transfer Elimination(UDTE) 기법을 제시하며, 본 논문의 프레임워크는 해당 알고리즘을 사용한다. UDTE의 핵심은 타겟에 오프로딩되는 메모리 영역의 접근 권한을 런타임에 제어하는 것이다.

표 9를 예로 들면 다음과 같다. 먼저, 첫 번째 커널을 실행하기 전에 buf[0:n] 영역에 설정된 CPU의 메모리 영역의 접근 권한을 박

탈한다. 이후 커널 실행과 메모리 복사 연산(GPU->CPU)을 스트림에 넣은 뒤 호스트(CPU)는 블록킹하지 않고 실행을 이어나간다. 만약, foo 함수가 CPU 상에서 buf[0:n] 영역을 수정하려고 할 경우 메모리 폴트가 발생하며, 이 경우에만 메모리 복사 연산을 기다린다. 만약 foo 함수가 buf[0:n]에 접근하지 않는다면 두 번째 루프 진입점에서 buf[0:n] 영역은 이미 디바이스에 맵핑되어 있으므로 메모리 복사(CPU->GPU)를 할 필요가 없어진다. 뿐만 아니라 UDTE는 불필요한 동기화 연산을 없애기도 하기 때문에 CPU-GPU 연산 간 overlapping을 가능해져 성능 향상을 가져온다.

제 7장 실험 결과 및 분석

본 장에서는 실험 환경을 제시하고, spec-accel 1.2 벤치마크를 이용한 실험 결과를 분석한다. 비교 대상은 순차 코드와 gcc7 컴파일러와 libgomp 런타임 시스템을 사용한 프로그램이며, 실험 결과 본 논문이 제시한 프레임워크는 평균적으로 순차 코드 대비 53.4배, gcc7 대비 6.75배 빠른 성능을 보였다. 나아가 제 7장에서는 OpenMP device constructs의 문제와 한계도 분석한다.

7.1 실험 환경

CPU	Intel Xeon Gold 6130 CPU 2.10GHz x 2 16 physical cores per CPU (Total 32 cores)
Memory	384 GB
GPU	Nvidia Tesla V100 PCI-e 16GB
OS	Ubuntu 16.04
Back-end Compiler	nvcc 9.1

표 1 실험 환경

실험은 Ubuntu 16.04 운영체제, NVIDIA Tesla V100로 구성된 시스템에서 수행되었다. 컴파일러를 거쳐 생성된 CUDA C 프로그램은 clang9 컴파일러를 이용하여 최종 CUDA 바이너리를 생성하였다. 비교 대상으로는 gcc7 컴파일러를 거쳐 생성된 CUDA 바이너리 파일을 사용하였다.

실험 대상은 SPEC ACCEL[6] 1.2에 포함되어 있는 stencil, lbm, mriq, ep, cg, sp, bt 총 7개의 C로 작성된 OpenMP C 프로그램을 사용하였다. 자세한 사항은 [6]을 참조하라.

실험은 OpenMP 기능을 사용하지 않은 serial 버전, gcc7을 사용하여 CUDA 바이너리를 생성한 버전과 비교하였다. 아래 절에서 본 논문의 프레임워크는 srcomp로 나타내었다.

7.2 실험 결과

7.2.1 gcc7과의 비교

실험 결과는 표 10, 표 11 및 그림 12와 같다. serial, gcc7과 srcomp는 각각 순차 코드, gcc 버전 7과 본 연구에서 제시된 프레임워크를 나타낸다. gcc7-simd는 모든 *parallel for* 프라그마를 *parallel for simd* 프라그마로 바꾼 뒤 gcc7을 이용해 컴파일한 버전의 성능을 나타낸다.

표 10은 실행 시간, 표 11은 순차 버전 대비 스피드업, 그리고 그림 12는 스피드업을 시각적으로 표현한 도식을 나타낸다. 실험은 모든 벤치마크를 10번씩 실행하여 진행하였으며, 표 10의 실행시간은 평균을 나타낸다. 표 11의 스피드업은 평균 실행 시간을 기준으로 산출되었으며, 그림 12의 gcc7-opt 스피드업은 gcc7과 gcc7-simd의 스피드업 중 높은 것을 선택하여 나타내었다.

gcc7과 gcc7-simd의 경우 알 수 없는 버그로 인해 mriq 벤치마크 실행에 실패하였다. 이에 따라 스피드업을 1로 설정하였다. geomean 항목은 7개 벤치마크의 스피드업 기하평균을 나타낸다. w/o mriq 항목은 mriq를 제외한 6개 벤치마크의 기하평균을 나타낸다. 실험 결과 srcomp가 gcc7 대비 6배 이상, mriq를 제외한 경우 2배 이상 뛰어난 성능을 보여준다는 것을 확인할 수 있다.

	Execution time(sec)			
	serial	gcc7	gcc7 -simd	srcomp
stencil	1093.3	190.3	95.7	7.9
lbm	856.0	67.6	67.6	31.4
mriq	13964.9			4.2
ep	7824.3	168.9	168.9	67.7
cg	1045.4	62.3	62.3	65.1
sp	1102.0	275.8	275.8	97.9
bt	685.4	209.7	164.9	143.8

표 2 vs gcc7 실행 시간

	Speedup			
	serial	gcc7	gcc7 -simd	srcomp
stencil	1	5.7	11.4	136.8
lbm	1	12.6	12.6	27.3
mriq	1	1	1	3543.0
ep	1	46.3	46.3	115.4
cg	1	16.7	16.7	16.1
sp	1	3.9	4.0	11.2
bt	1	3.2	4.1	4.7
geomean	1	6.89	7.87	53.4
w/o mriq	1	9.5	11.1	26.8

표 3 vs gcc7 스피드업 수치

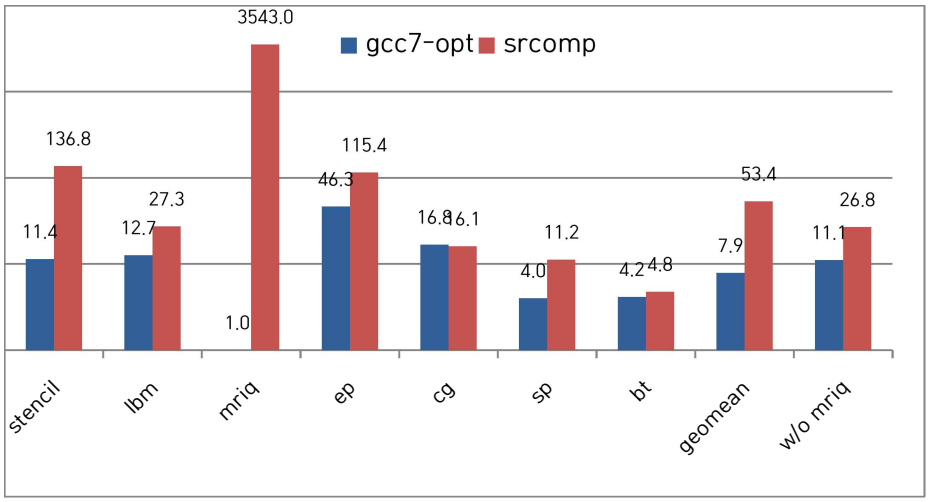


그림 12 vs gcc7-opt 스피드업 차트

7.2.2 PGI OpenACC과의 비교

spec-accel 1.2 벤치마크는 OpenMP 뿐만 아니라 OpenACC으로 작성된 벤치마크도 제공한다. OpenACC은 컴파일러 프라그마 기반으로 GPU 상에서 프로그램을 실행하게 하는 표준이며, pgi에서 상용 컴파일러를 제공하고 있다[8]. 본 절에서는 pgi 컴파일러와 본 논문의 프레임워크를 비교 분석한다.

OpenMP와 OpenACC이 동일한 벤치마크 프로그램들을 지원하지만 내부 구현과 시맨틱은 다르다는 점에 주의해야 한다. 따라서 본 절의 실험 결과는 직접적인 프레임워크의 성능을 측정하기 보다는 OpenMP 프레임워크의 개선 여지를 확인하는 데 의의가 있을 것이다.

표 4, 표 5, 그림 18은 실험 결과를 나타낸다. 각 항목은 7.2.1에서 정의한 바와 같으며, OpenACC 항목은 pgi 컴파일러를 사용했을 때의 성능을 나타낸다. 실험 결과 OpenACC이 본 프레임워크보다 2배 이상 높은 성능을 보였다.

이러한 차이는 OpenACC과 OpenMP 시맨틱의 미묘한 차이와 컴파일러의 최적화 능력에서 나타났다. 예컨대, OpenACC은 *acc kernel loop* 프라그마를 사용하면, 컴파일러 분석을 통해 병렬화할 수 있는 루프를 특정하고, 최대한 많은 루프를 확보하여 병렬화한다. 반면, OpenMP는 collasped clause를 사용해서 사용자가 직접 루프를 특정해야 한다. 이중 루프에서 외부 루프의 코드가 side-effect가 없는 경우를 예로 들자. OpenACC의 경우 컴파일러 최적화를 통해 내부 루프와 외부 루프를 coallpse할 수 있지만, OpenMP는 코드 수정 없이 불가하다.

	Execution time(sec)		
	serial	pgi	srcomp
stencil	1093.388	8.6	7.992
lbm	856.05	39.9	31.406
mriq	13964.97	3.9	4.206
ep	7824.318	52.7	67.786
cg	1045.466	5.9	65.136
sp	1102.02	18.4	97.9675
bt	685.48	8.6	143.864

표 4 vs pgi 실행 시간

	Speedup		
	serial	pgi	srcomp
stencil	1093.388	126.01	136.8
lbm	856.05	21.4	27.3
mriq	13964.97	3543.03	3543.0
ep	7824.318	148.27	115.4
cg	1045.466	176.7781535	16.1
sp	1102.02	59.801	11.2
bt	685.48	79.96	4.7
geomean	1	142.39	53.4

표 5 vs pgi 스피드업

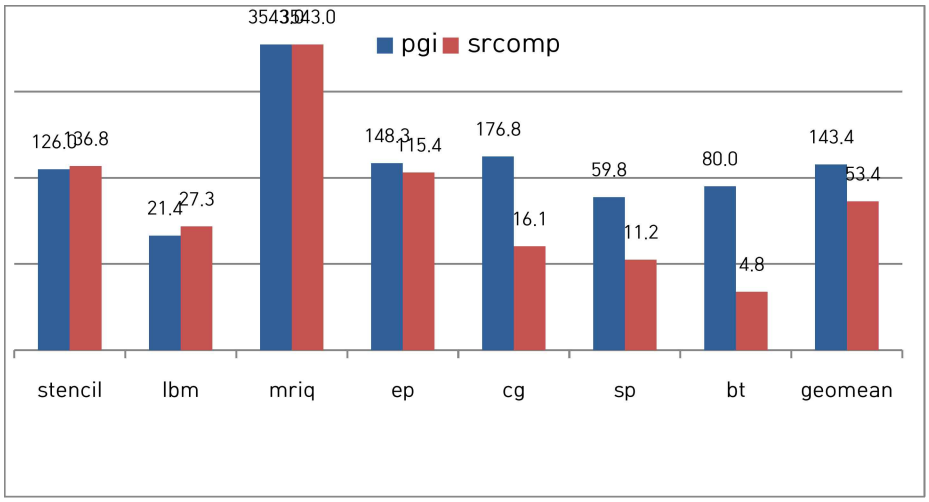


그림 17 vs pgi 스피드업

7.2.3 논의

spec-accel 벤치마크의 경우 최적화된 코드로 구성되어 있어서, 실제 불필요한 연산 및 통신이 거의 없고, 따라서 런타임 최적화 기법을 통한 성능 향상을 확인하기 쉽지 않았다. 본 연구는 추가적인 벤치마크를 활용하여 보다 자세한 실험 및 분석으로 확장될 수 있을 것으로 기대된다.

한편, 본 논문의 프레임워크가 gcc7 대비 뛰어난 성능을 보인 것은 크게 두 가지 원인에서 기인한다. 첫 번째는 gcc7은 srcomp와 달리 자체적으로 CUDA 바이너리를 생성한다. gcc7의 CUDA 프로그램 컴파일 기능은 최근에 지원되기 시작했으므로 srcomp 프레임워크가 백엔드로 사용하는 clang, nvcc 등의 컴파일러보다 최적화 기술이 현저히 떨어진다. 두 번째, gcc7의 경우 성능 이식성이 떨어진다. SPEC ACCEL 벤치마크 프로그램의 경우 대부분 성능 최적화를 위해 파라미터 값을 따로 설정해주지 않으며, 기본값을 사용하도록 한다. gcc7의 경우 이러한 파라미터들을 명시적으로 조정하거나 혹은 코드를 수정하여 다른 프라그마를 사용할 경우 성능 향상을 보였으며, 특히 stencil 벤치마크에서 5배 이상의 성능 차이가 나는 것을 확인할 수 있다. 반면, srcomp의 경우 파라미터 설정에 크게 영향을 받지 않았다.

제 7장 결론

본 논문은 OpenMP 4.5 device construct를 사용해 작성된 C 프로그램을 CUDA 프로그램으로 변환하는 프레임워크를 제시하였고, 실험 결과 gcc7 보다 향상된 성능을 보여주었다.

OpenMP와 CUDA의 실행 모델, 메모리 모델 및 동기화 과정을 조망하고, source-level 컴파일러의 디자인과 정확성을 보여주었다. 또한, 성능 향상을 위한 동적 구간 트리, 동적 커널 선택, 버디 할당자, UDTE와 같은 런타임 시스템 최적화 기술을 도입하였다.

spec-accel 1.2 벤치마크를 이용한 실험 결과 gcc7 대비 6배 이상, mriq를 제외한 경우 2배 이상의 성능을 보여주었다.

본 연구는 추가적인 벤치마크를 활용하여 보다 자세한 실험 및 분석으로 확장될 수 있을 것이다. spec-accel의 경우 비교적 최적화된 코드로 구성되어 있어서 실제 런타임 최적화 기법을 통한 성능 향상을 확인하기 쉽지 않았다. 추가적인 벤치마크를 활용하면 런타임 최적화 기법 별로 성능 향상 정도를 파악할 수 있을 것이다. 나아가 본 연구를 바탕으로 향후에는 추가적인 런타임 최적화 기능 및 컴파일러 최적화 기술을 적용할 수 있을 것으로 예측된다.

참고문헌

[1] OpenMP. OpenMP Application Programming Interface. OpenMP. Nov, 2015.

[2] NVIDIA. CUDA C Programming Guide. NVIDIA. Nov, 2015.

[3] Lee, Seyong, Seung-Jai Min, and Rudolf Eigenmann. "OpenMP to GPGPU: a compiler framework for automatic translation and optimization." *ACM Sigplan Notices* 44.4 (2009): 101-110.

[4] Kim, J., Lee, Y. J., Park, J., & Lee, J. Translating OpenMP device constructs to OpenCL using unnecessary data transfer elimination. In SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (pp. 597-608). IEEE. 2016.

[5] gcc. GCC 7 Release Series. 2018.

[6] Juckeland, Guido, et al. SPEC ACCEL: A standard application suite for measuring hardware accelerator performance. International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems. Springer, Cham, 2014.

[7] Knowlton, Kenneth C. "A fast storage allocator." *Communications of the ACM* 8.10 (1965): 623-624.

[8] *OpenACC 2.0a Specification*. June 2013.

Abstract

Translating OpenMP Device Constructs for NVIDIA GPUs

Park Daeyoung
Department of Computer Science & Engineering
The Graduate School
Seoul National University

This paper deals with a framework composed of a compiler and runtime system that converts C programs written using the OpenMP 4.5 device construct into CUDA programs. To this end, we present a compiler and runtime library that converts OpenMP C programs to CUDA programs. First, we look at the execution model, memory model, and synchronization process of OpenMP and CUDA, and explain how to secure the design and accuracy of the source-level compiler. In addition, runtime system optimization techniques such as dynamic section tree, dynamic kernel selection, buddy allocator, and UDTE are introduced to improve performance.

Using the spec-accel 1.2 benchmark, it showed more than 6 times the performance of gcc7, and more than 2 times when excluding mriq. It is expected that additional runtime optimization and compiler optimization techniques can be applied based on the framework of this paper.

Keywords : Heterogeneous System, OpenMP, CUDA, Compiler, GPU
Student Number : 2018-24786