SYCL에서 효율적인 멀티 GPU 프로그래밍을 위한 MPI-wrapper API 개발

명훈주, 구기범, 오광진
한국과학기술정보연구원 슈퍼컴퓨팅기술개발센터
hjmyung@kisti.re.kr, gibeom.gu@kisti.re.kr, koh@kisti.re.kr

Development of MPI-wrapper for efficient SYCL-based Multi GPU programming

Hunjoo Myung, Gibeom Gu, Kwang jin Oh
Center of Supercomputing Technology Development, Korea Institute Science and Technology Information

요 약

SYCL은 C++을 기반으로 하는 언어로 가속기를 사용하는 복잡한 과정을 C++의 특징 중의 하나인 추상화를 사용해 개발자가 쉽게 접근할 수 있게 한다. 그러나, 가속기를 활용하는 측면에서는 성능을 최대한으로 끌어내기 위해 저수준 접근도 필요하다. 특히, NVLink와 같이 효율적인 멀티-GPU 통신을 해주는 인터커넥션 링크 활용을 위해서도 필요하다. 본 논문에서는 SYCL 구현물 중의 하나인 AdaptiveCpp을 가지고 NVLink로 연동된 멀티 GPU 환경에서 효율적으로 프로그래밍을 할 수 있는 방법을 제안하고, SYCL 개발자들이 SYCL의 설계 철학을 따라 프로그래밍을 할 수 있도록 이러한 기능을 추상화하여 담은 MPI wrapper API를 제안한다.

1. 서론

SYCL은 OpenCL과 같이 Khronos Group[1]에서 만들고 있는 표준 C++프로그래밍 모델로서, 특정 벤더의 디바이스에 종속되지 않고, 다양한 디바이스를 지원하고 있다. 또한 SYCL은 C++기반으로 하기 때문에, C++의 강력한 기능을 그대로 물려받고 있다. 즉, C++의 강력한 기능에는 추상화를 비롯한 여러 객체 지향 언어의 특징이 있는 반면에, C와 유사하게 빠른 성능을 위한 저수준의 프로그래밍도 제공하는데, SYCL 역시 이러한 특징을 물려받았다.

SYCL이 가속기를 비롯한 여러 디바이스를 다루는 프로그래밍 모델이므로 기본적으로 빠른 성능을 갖추어야 한다. 아울러 최근 슈퍼컴퓨터들은 대부분 GPU 기반이므로, 이런 환경에서 SYCL이 개발 도구로 사용되려면, 멀티 GPU 프로그래밍이 가능해야 한다. 본 논문에서는 SYCL로 빠른 성능을 내는 멀티 GPU 프로그래밍 방법에 대해 고찰하고 이를 위한 MPI wrapper를 제안한다. 2장에서는 멀티 GPU 환경에서 SYCL의 한계에 대해 서술하고 3장에서는 MPI 기반의 SYCL 프로그래밍 모델에 대해 제안한다. 그리고 4장에서는 이를 위한 MPI wrapper를 제안하고 5장에서 결론을 맺는다.

2. 멀티 GPU 환경에서의 SYCL 한계

2.1 SYCL의 메모리 관리

SYCL은 디바이스와 호스트 간의 메모리 관리를 추상화시킨 sycl::buffer와 sycl::accessor 등의 클래스를 제공한다. 이 클래스들을 이용하면 개발자는 디바이스와 호스트 간의 메모리 동기화에 대해서는 고려하지 않고, <그림 1,2>와 같은 방법으로 디바이스 혹은 호스트에서 데이터 접근이 가능하다. 즉, SYCL에서는 가속기와 호스트 간의 메모리 동기화를 기본적으로 제공한다.

```
std::vector v;
...
sycl::buffer buf(v.data(), v.size());
queue.submit([&](sycl::handler& cgh){
  auto my_v= buf.get_access<...>(cgh);
  cgh.parallel_for(...);
}
```

<그림 1> SYCL 디바이스에서의 데이터 접근

```
std::vector v;
...
sycl::buffer buf(v.data(), v.size());
auto my_v = buf.get_access<...>();
```

<그림 2> SYCL 호스트에서의 데이터 접근

이러한 SYCL의 데이터 접근 방법은 개발자에게 데이터 동기화에 대한 고려가 필요 없는 쉬운 프로그래밍을 제공하지만, 최적 성능을 끌어내야 하는 측면에서는 단점들이 존재한다. 가장 큰 단점 중의 하나는 메모리 관리의 추상화로 인해 sycl::buffer 객체에서 GPU의 메모리 포인터를 얻을 수 있는 방법이 존재하지 않는다. 이로 인해 SYCL에서는, NVIDIA GPU를 예로 들면, NVIDIA에서 제공하고 있는 cublas[2], cufftw[3] 등의 최적화된 라이브러리를 사용할 수 있는 방법이 없다.

2.2 SYCL에서의 멀티 GPU 프로그래밍

SYCL에서 멀티-GPU 프로그래밍의 방법은 각 GPU마다 sycl::queue 객체를 생성하고 명시적으로 데이터를 분할하여 각 sycl::queue에 작업을 제출하는 방법, MPI 프로그래밍, 그리고 Celertiy[4] 등과 같이 앞서 언급한 멀티 GPU 프로그래밍을 추상화하여 API를 제공하는 솔루션을 이용하는 방법 등 세 가지 방법이 있다. 본 논문은 SYCL에서 보다 빠른 성능의 멀티 GPU 프로그래밍 관점으로 접근하고 있으므로, 세 번째 방법은 제외한다.

멀티 GPU 프로그래밍의 경우, GPU 간의 효율적인 데이터 전송이 가장 중요한 요소 중의 하나이다. 이것을 가능하게 하려면, 앞에서 언급한 첫 번째와 두 번째 방법 모두 GPU 메모리 포인터를 가지고 저수준 프로그래밍을 해야 한다. 이 외에도 첫 번째 방법의 경우에는 GPU 벤더에서 제공하는 피어 디바이스 메모리 접근 (Peer Device Memory Access) API [5][6]등을 이용해야 하고, 두 번째의 경우에는 GPU-aware MPI를 이용해야 한다. 본 논문에서는 두 번째 경우에 대해 한정하여 구체적으로 설명한다.

3. MPI 기반의 효율적인 SYCL 프로그래밍 모델

3.1 AdaptiveCpp[7]에서의 구현 방법

앞서 언급했듯이, sycl::buffer 객체에서 GPU 메모리 포인터를 얻는 방법은 기본적으로 존재하지 않는다. 다만, SYCL 구현물마다 SYCL 명세 이외에도 확장 API를 제공한다. AaptiveCpp의 경우에는 sycl::handler::hipSYCL_enqueue_custom_operation() 메소드를 제공하는데, 이를 통해서 <그림 4>과 같이 GPU 메모리 포인터를 얻을 수 있고, 이 포인터를 매개변수로 받는 GPU 벤더가 제공하는 API나 라이브러리도 사용가능하다.

```
sycl::queue q;
q.submit([&](sycl::handler &cgh){
  auto acc=A_buff.get_access<...>(cgh);
  cgh.hipSYCL_enqueue_custom_operation(
    [=](sycl::interop_handler &h){
      void *native_mem=
       h.get_native_mem<sycl::backend::cuda>(acc);
       // <-GPU 메모리 포인터를 얻는 방법
       ...(중략)...
    });
});
```

<그림 3> AdaptiveCpp에서 GPU 메모리 포인터를 얻는 방법

3.2 AdaptiveCpp에서의 효율적인 MPI 프로그래밍 방법

앞서 언급했듯이, MPI를 이용해 멀티 GPU 프로그래밍을 할 때, 효율적인 GPU 간의 데이터 전송을 위해 GPU 메모리 포인터와 GPU-aware MPI를 사용해야 한다. 이것은 GPU-aware MPI을 사용하기 위한 문법적인 관점이고, GPU 간의 데이터 전송 등의 효율적인 성능은 GPU-aware MPI 구현물이 제공한다. 즉, MVAPICH2 GDR [8]과 OpenUCX [9]와 연동한 OpenMPI의 경우, 노드 간은 GPUDirect [10]와 같은 효율적인 전송 계층을 지원하고, 노드 안에서는 NVLink [11]와 같은 대역폭이 높은 링크를 이용하는 CUDA IPC [12] 등을 지원하는데, 이를 사용해 성능을 끌어낼 수 있다.

3.2 MPI 기반 SYCL 성능 평가

본 절에서는 앞 절에서 언급한 성능향상을 위한 MPI 기반 멀티-GPU 프로그래밍 방법에 대한 성능 평가를 수행하였다. 실험 환경은 아래와 같다.

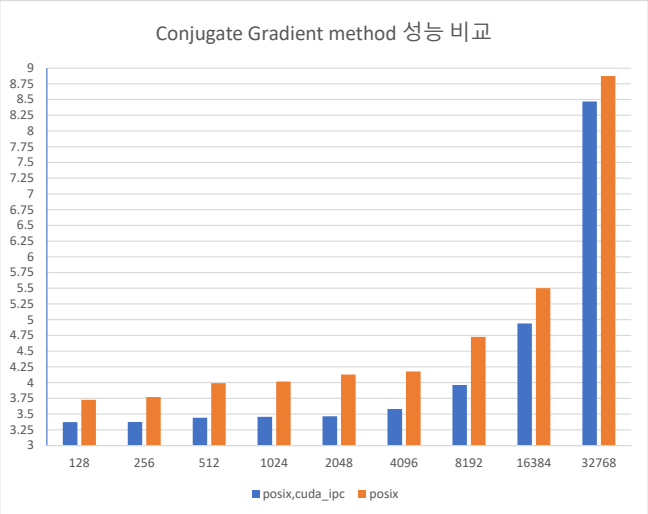
[실험 환경]

- 시스템: NVIDIA DGX A100
  - CPU: AMD EPYC 7742 64-Core Processor 1개
  - 주메모리: 512 GB
  - GPU: NVIDIA Tesla A100 40GB x 4개
  - GPU 인터커넥트: NVIDIA NVSwitch fabrics

- OS: 우분투 20.04
- MPI: OpenUCX와 연동한 OpenMPI 4.1.5

벤치마크 코드는 희소행렬 방정식을 위한 CG (conjugate gradient) 메소드를 SYCL로 직접 작성하였다. MPI와 CUDA-aware MPI을 각각 사용하는 두 가지 버전으로 작성하였으며, 작업 수행 시에 MPI를 사용하는 CG 코드는 POSIX 공유 메모리를 사용하여 노드 내의 4개 프로세스가 통신하게 설정하였고, CUDA-aware MPI를 사용하는 CG 코드는 POSIX 공유메모리와 CUDA IPC를 사용하여 4개의 프로세스가 통신하게 설정하였다 (환경변수를 각각 UCX_TLS=posix, UCX_TLS=posix,cuda로 설정하였다).

실험조건은 희소행렬 128 x 128부터 32,768 x 32,768까지 크기를 증가시키면서 수행시간을 측정하였고, 실험 결과는 <그림 4>와 같다. CUDA-aware MPI를 사용한 코드는 MPI를 사용한 코드에 비해 4~19%까지 성능향상을 보였다.



<그림 4> NVLINK로 연결된 멀티 GPU 환경에서의 CG 성능 평가

4. SYCL을 위한 MPI Wrapper 설계

앞 절에서 서술한 바와 같이, GPU 메모리 포인터와 GPU-aware MPI를 이용한 CG 코드가 MPI를 이용한 것보다 좋은 성능을 보여주지만, SYCL의 설계 철학에 반하는 접근방법이다. 이 문제점을 극복하기 위해서 우리는 MPI Wrapper를 설계하였다. 이 Wrapper를 통해 얻는 효과는 다음과 같은 두 가지이다. 첫째, SYCL 개발자는 디바이스 메모리 포인터를 이용하여 프로그래밍하는 부분을 API 호출함으로써, 사용자는 디바이스 포인터를 직접 다루지 않고 SYCL의 설계 철학을 따라 프로그래밍을 할 수 있다. 둘째, MPI와 유사한 인터페이스를 제공함으로써, SYCL에서 보다 쉽고 성능이 좋은 MPI 프로그래밍을 가능하게 해준다. 아래 <그림 5>은 MPI_Send API에 대응하는 Wrapper의 API이다.

```
mpi_send(sycl::queue q, sycl::buffer src,
         int count, sycl::buffer dest,
         int tag,  MPI_Comm comm){
 q.wait(); // 동기화
 // src의 데이터타입을 확인하는 코드 (생략)
q.submit([&](sycl::handler &cgh){
  auto s= src.get_access<sycl::access::mode::read>(cgh);
  auto d=dest.get_access<sycl::access::mode:write>(cgh);
  cgh.hipSYCL_enqueue_custom_operation(
      [=](sycl::interop_handler &h){
       void *s_mem=
        h.get_native_mem<sycl::backend::cuda>(s);
       void *d_mem=
        h.get_native_mem<sycl::backend::cuda>(d);
       MPI_Send(s_mem, count, data_type, d_mem,...);
      });
 });
 q.wait();
}
```

<그림 5> AdaptiveCpp에서 GPU 메모리 포인터를 얻는 방법

5. 결론

SYCL은 CUDA와는 달리 다양한 디바이스를 지원한다는 것과 C++의 장점을 이어받아 OpenCL보다 한층 더 높은 추상화를 제공해 개발자에게 보다 쉬운 프로그래밍을 제공한다. 그러나, SYCL이 CUDA나 HIP과 같은 벤더에서 제공하고 있는 개발도구를 뛰어넘으려면, 최적의 성능을 끌어낼 수 있는 저수준 프로그래밍도 가능하게 해야 한다. 아울러, GPU 클러스터 혹은 GPU 기반 슈퍼컴퓨터를 이용해 대규모 작업을 수행할 때는 디바이스 간의 통신 오버헤드를 줄이는 것이 필수적이고, 우리가 제안한 MPI wrapper는 이러한 하드웨어를 사용하는 SYCL 개발자들에게 좋은 해결책이 될 것이다.

이 논문은 대한민국 정부(과학기술정보통신부)의 재원으로 한국과학기술정보연구원 초고성능컴퓨팅 공동활용을 위한 통합 환경 개발 및 구축 사업과 한국연구재단 슈퍼컴퓨터개발선도사업의 지원을 받아 수행된 연구임 (과제번호: 2020M3H6 A1084857)

참고문헌

[1] https://www.khronos.org/
[2] https://developer.nvidia.com/cublas
[3] https://docs.nvidia.com/cuda/cufft/index.html
[4] https://celerity.github.io/
[5] https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__PEER.html
[6] https://docs.amd.com/projects/HIP/en/docs-5.2.0/doxygen/html/group___peer_to_peer.html
[7] https://github.com/AdaptiveCpp/AdaptiveCpp
[8] https://mvapich.cse.ohio-state.edu/userguide/gdr/
[9] https://openucx.org/
[10] https://developer.nvidia.com/gpudirect
[11] https://www.nvidia.com/ko-kr/data-center/nvlink/
[12] https://openucx.org/wp-content/themes/jello/uploads/UCX_SC18_BOF_NVIDIA_OpenMPI-UCX-CUDA-on-DGX.pdf