GPU 메모리의 데이터 중복 적재를 방지하기 위한 GPU 프로세스간 데이터 공유 기법 연구

강지훈*
한국방송통신대학교 컴퓨터과학과 교수

Data sharing techniques between GPU processes to avoid duplicate loading of data in GPU memory

Jihun Kang*
Professor, Department of Computer Science, Korea National Open University

요 약 단일 GPU(Graphics Processing Unit)에서 여러 프로세스가 동일한 읽기 전용 데이터를 개별 적재하면 GPU 메모리 사용량과 H2D(Host-to-Device) 전송 비용이 증가한다. 본 연구는 중앙 관리 프로세스가 등록된 공용 데이터를 한 번 적재하고 CUDA IPC(Inter-Process Communication)로 공유하는 기법을 제안한다. 선택적 API(Application Programming Interface) 중재는 동일한 cudaMemcpy() 호출을 객체 상태와 프로세스 역할에 따라 최초 적재 또는 재사용 확인으로 처리한다. 행렬-벡터 곱셈 실험에서 공용 데이터의 논리적 GPU 메모리 사용량은 프로세스 2개와 4개에서 각각 50%와 75% 감소하였다. 네 용량 경계 조건의 20쌍 비교에서 독립 적재 방식은 모두 GPU 메모리 부족으로 실패했으나 제안 방식은 모두 완료하였다. 공용 데이터가 적재된 상태에서 256 MiB, 512 MiB 및 1 GiB의 후속 GPU 준비 시간은 각각 95.74%, 97.93% 및 98.92% 감소하였다. 커널 성능 차이는 미미했으며 계산 결과 불일치는 없었다.

주제어 : GPU 메모리 공유, 다중 프로세스 컴퓨팅, CUDA IPC, API 중재, 데이터 재사용

Abstract When processes load identical read-only data onto one graphics processing unit (GPU), memory use and host-to-device transfer costs increase. We propose loading registered data once through a central manager and sharing them via CUDA inter-process communication (IPC). Selective application programming interface (API) interposition handles the same cudaMemcpy() call as initial loading or reuse confirmation based on object state and process role. In matrix-vector multiplication experiments, logical common-data memory use decreased by 50% and 75% with two and four processes, respectively. Across four capacity-boundary configurations, independent loading failed with out-of-memory errors in all 20 paired trials, whereas the proposed method completed all workloads. With data resident, subsequent preparation time decreased by 95.74%, 97.93%, and 98.92% for 256 MiB, 512 MiB, and 1 GiB. Kernel performance remained comparable, with no result mismatches.

Key Words : GPU Memory Sharing, Multi-Process Computing, CUDA IPC, API Interposition, Data Reuse

이 논문은 2024년도 4분기 한국방송통신대학교 학술연구비 지원을 받아 작성된 것임
*교신저자 : 강지훈(k2j23h@knou.ac.kr)
접수일 2026년 08월 26일 수정일 2026년 09월 14일 심사완료일 2026년 10월 16일

www.kci.go.kr

1. 서론

GPU(Graphics Processing Unit)는 높은 병렬 처리 성능을 제공하지만 장치 메모리 용량은 제한되어 있다. CUDA(Compute Unified Device Architecture) 환경의 독립 프로세스는 각각 실행 문맥과 주소 공간을 사용하므로, 동일한 모델 가중치나 상수 데이터를 사용하더라도 GPU 메모리 할당과 H2D(Host-to-Device) 전송을 반복한다[1,2]. 이에 따라 공용 데이터의 논리적 GPU 메모리 사용량은 데이터 크기와 프로세스 수에 비례하여 증가하며, GPU 연산 자원에 여유가 있어도 메모리 부족으로 동시 실행 수가 제한될 수 있다. 이 문제는 장치 확장이 어렵고 동일 데이터를 반복 사용하는 단일 GPU 엣지 노드에서 더욱 중요하다.

기존 연구는 GPU 공간 공유와 작업 스케줄링[3,4,5], 엣지 환경의 GPU 공유[6], GPU 메모리 관리와 동시 학습 성능 예측[7,8,9], MPS(Multi-Process Service)와 MIG(Multi-Instance GPU) 기반 공간 공유 특성[10]을 다루었다. 이들 연구는 주로 자원 분할, 작업 배치, 메모리 이동 또는 간섭 예측을 다룬다. 본 연구는 등록된 읽기 전용 공용 객체를 공유 단위로 하여 동일 호스트의 독립 CUDA 프로세스 간에 하나의 GPU 메모리 할당을 재사용하고, 메모리 호출을 최초 적재와 재사용 확인으로 구분하여 중복 할당과 반복 H2D 전송을 줄인다.

본 연구는 중앙 관리 프로세스와 선택적 API(Application Programming Interface) 중재를 결합한 GPU 데이터 공유 기법을 제안한다. 중앙 관리 프로세스는 공유 객체의 원본 GPU 메모리와 상태·참조 수를 관리하며, 최초 적재 프로세스가 공유 객체의 기본 포인터를 대상으로 호출한 전체 크기의 H2D cudaMemcpy()는 실제 데이터 적재로 처리한다. 재사용 프로세스의 동일 호출은 CUDA IPC(Inter-Process Communication)로 연결된 기존 객체의 적재 완료와 사용 가능 상태를 확인하는 경로로 전환한다. 적용 시 클라이언트 라이브러리를 통한 공유 객체의 명시적 요청·등록, 동적 CUDA Runtime 연결 및 LD_PRELOAD 설정이 필요하다. 중재 모듈은 cudaMemcpy()와 cudaFree() 호출을 조건에 따라 재지정·통과·거부하며, 등록된 객체에 대한 기존 호출 형식과 프로세스별 GPU 커널·전용 버퍼 관리 구조는 유지한다.

본 연구의 기여는 다음과 같다. 첫째, 동일한 cudaMemcpy() 호출을 객체 상태와 프로세스 역할에 따라 최초 적재와 재사용으로 구분하는 상태 기반 선택적 API 재지정 기법을 제안하였다. 둘째, 중앙 관리 프로세스, 클라이언트 라이브러리, LD_PRELOAD 중재 모듈, CUDA IPC 연결 및 참조 기반 생명주기를 결합한 프로토타입을 구현하였다. 셋째, 동일한 실행 파일, 데이터 및 GPU 커널을 사용하는 행렬-벡터 곱셈 기반 통제 실험으로 GPU 메모리 사용량, 동시 실행 용량, 후속 데이터 준비 시간, 커널 성능, 계산 정확성 및 프로세스 종료 후 자원 회수를 평가하였다.

2. 연구 배경

2.1 프로세스별 GPU 메모리와 논리적 중복

CUDA Runtime에서 cudaMalloc()으로 할당한 포인터는 해당 프로세스의 CUDA 실행 문맥에서만 유효하므로, 동일한 읽기 전용 데이터를 사용하는 독립 프로세스도 각각 GPU 메모리를 할당하고 데이터를 전송한다[1,2]. 본 연구는 공유 대상으로 등록한 읽기 전용 공용 데이터의 내용이 동일하다는 전제에서 하나의 GPU 메모리 할당을 공유하며, CUDA 실행 문맥과 입력·출력·임시 버퍼는 프로세스별로 유지한다.

동일한 공용 데이터를 사용하는 프로세스 수를 N, 공용 데이터 크기를 D라고 하면 독립 적재 방식의 공용 데이터 사용량은 N×D, 제안 방식은 D이며 논리적 절감량은 (N-1)×D이다. 실제 총 GPU 메모리에는 배경 사용량, 작업 프로세스별 CUDA 실행 문맥과 전용 버퍼가 포함되며, 제안 방식에는 중앙 관리 프로세스의 CUDA 실행 문맥에 따른 고정비용도 추가된다. 따라서 공용 데이터 중복분의 절감량이 관리자 고정비용보다 작으면 논리적 공용 데이터 사용량이 감소하더라도 총 GPU 메모리는 증가할 수 있다. 본 연구에서는 공용 데이터의 논리적 GPU 메모리 사용량과 NVML (NVIDIA Management Library)이 측정한 총 GPU 메모리 사용량을 구분하여 분석한다[11].

2.2 CUDA IPC와 선택적 API 중재

CUDA IPC는 동일한 호스트와 물리 GPU를 사용하는 프로세스가 하나의 GPU 메모리 할당을 공유하도록 지원한다[1,2]. 원본 할당 프로세스는 cudaIpcGetMemHandle()로 핸들을 생성하고, 다른 프로세스는 cudaIpcOpenMemHandle()로 자신의 CUDA 실행 문맥에 해당 메모리를 연결한다. 각 프로세스는 사용 종료 시 cudaIpcCloseMemHandle()로 매핑을 닫아야 하며,

www.kci.go.kr

원본 할당은 모든 매핑이 종료된 후 소유 프로세스가 해제한다. CUDA IPC는 프로세스 간 GPU 메모리 연결을 제공하며, 공유 키 기반 객체 식별, 최초 적재 역할 결정, 중복 전송 생략 및 상태·참조 관리는 제안 시스템에서 별도로 수행한다.

Linux 동적 링커는 LD_PRELOAD에 지정된 공유 라이브러리를 우선 적재하며, 중재 함수는 dlsym()과 RTLD_NEXT를 이용하여 원래 함수 심볼을 호출할 수 있다[12,13]. 본 연구의 작업 프로세스는 동적 CUDA Runtime인 libcudart.so.12에 연결하고, 중재 모듈은 cudaMemcpy()와 cudaFree()만 선택적으로 처리한다. 비동기 전송과 스트림별 중재는 현재 구현 범위에 포함하지 않는다. 공유 객체와 관련되지 않은 호출은 기존 CUDA Runtime으로 전달하며, 공유 호출의 재지정·통과·거부 조건은 4.3절에서 설명한다.

3. 관련 연구

GPU 공간 공유와 작업 스케줄링 연구는 GPU 자원 배분과 작업 간 간섭 완화에 중점을 둔다. MuxFlow는 온라인·오프라인 작업을 하나의 GPU에 공간적으로 배치하고 메모리와 연산 자원을 보호하며[3], InSS와 FineST는 추론 작업의 공간·시간 공유를 통해 처리량을 높이면서 지연 시간 요구를 고려한다[4,5]. Villarrubia 등은 MPS와 MIG 기반 동시 실행의 성능, 에너지 사용량 및 메모리 경합 특성을 비교하였다[10]. 이들 연구는 GPU 자원의 분할·배치와 실행 간섭을 주요 대상으로 하며, 독립 프로세스가 동일한 읽기 전용 데이터를 하나의 GPU 메모리 할당으로 공유하는 방식은 다루지 않는다.

Woo 등은 엣지 장치에서 스레드 방식과 다중 프로세스 방식의 메모리 사용 및 GPU 활용 특성을 비교하였다[6]. 스레드 방식은 동일 프로세스의 CUDA 실행 문맥과 메모리를 공유하지만, 본 연구는 독립적인 CUDA 실행 문맥을 유지하는 프로세스 간 공용 객체 공유를 대상으로 한다. vTensor는 CPU와 GPU 간의 텐서 교환을 세분화하여 전송 오버헤드를 줄인다[7]. TensorShare는 동시 학습 작업의 완료 시간 증가를 예측하여 작업 조합을 배치하고[8], Chen 등은 MPS 환경의 연산·메모리 간섭과 실행 시간을 예측한다[9].

대표적인 GPU 공유·메모리 관리 접근과 제안 기법을 공유 단위, 프로세스 독립성, 응용 연동, CUDA IPC 활용, 메모리 절감 방식, 반복 H2D 전송 생략 및 적용 제약에 따라 비교한 결과는 〈Table 1〉과 같다.

본 연구는 CUDA IPC의 메모리 연결 기능에 공유 키 기반 객체 식별, 객체별 최초 적재·재사용 역할 및 참조 생명주기 관리를 결합한다. 공유 객체 요청·등록을 위한 응용 연동은 필요하지만, 등록된 객체에 대한 기존 cudaMemcpy()·cudaFree() 호출 형식과 프로세스별 GPU 커널·전용 버퍼 구조는 유지한다. 유효한 전체 객체 H2D 호출을 객체 상태와 프로세스 역할에 따라 최초 적재 또는 재사용 확인으로 처리함으로써, 독립 CUDA 프로세스의 실행 구조를 유지하면서 공용 데이터의 중복 할당과 반복 전송을 함께 줄이는 것이 본 연구의 강점이다.

〈Table 1〉 Comparison of GPU Sharing and Memory Management Approaches

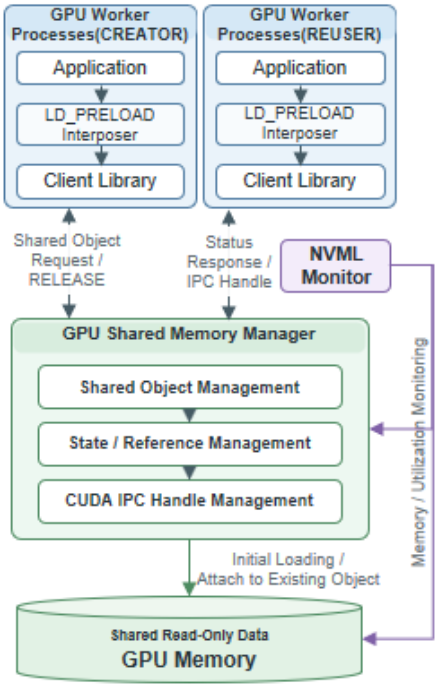
Study	Sharing / Management Unit	Independent Processes	Integration	CUDA IPC	Memory Effect	H2D Reuse	Main Constraints
MuxFlow [3]	GPU resources	Yes	Runtime hooks	Not confirmed from available sources	Quota control	Not targeted	MPS / interference
InSS [4]	GPU resources	Not confirmed from available sources	Scheduler	〃	Not targeted	〃	Multi-GPU inference
FineST [5]	GPU resources	〃	Scheduler	〃	〃	〃	Inference latency
Woo et al. [6]	Context / data	Mode-dependent	Threads / processes	〃	Context sharing	〃	Edge / contention
vTensor [7]	Sub-tensors	Not targeted	Memory layer	〃	Offloading	〃	Training / swapping
TensorShare [8]	Training jobs	Not confirmed from available sources	Scheduler	〃	Not targeted	〃	Prediction / SLA
Chen et al. [9]	Training tasks	Yes	Profiling	〃	〃	〃	MPS / profiling
Villarrubia et al. [10]	GPU resources	Yes	Configuration	〃	〃	〃	Hardware / mode
Direct CUDA IPC [1,2]	Allocation	Yes	IPC code	Yes	Shared allocation	Application-managed	Lifetime control
Proposed	Read-only object	Yes	Registration + hooks	Yes	Shared allocation	State-based	Same GPU / API limits

www.kci.go.kr

4. 제안 기법 및 구현

4.1 전체 구조

제안 시스템은 중앙 관리 프로세스, 클라이언트 라이브러리, LD_PRELOAD 중재 모듈, GPU 작업 프로세스 및 독립 모니터링 프로세스로 구성된다. 작업 프로세스는 클라이언트 라이브러리를 통해 공유 객체를 요청·연결하고, 프로세스별 전용 버퍼를 준비하여 GPU 커널을 실행한다. LD_PRELOAD 중재 모듈은 cudaMemcpy()와 cudaFree() 호출을 공유 포인터와 호출 조건에 따라 재지정·통과·거부한다. 중앙 관리 프로세스는 원본 GPU 메모리를 소유하며, 공용 데이터의 할당·최초 적재·해제와 공유 객체의 상태·참조 수 관리를 담당한다. 독립 모니터링 프로세스는 NVML로 GPU 메모리와 활용률을 수집하며, 공유 객체의 상태·참조 수 변경이나 자원 회수에는 관여하지 않는다. 전체 구성과 데이터 흐름은 [Fig. 1]에 나타낸다.



[Fig. 1] Overall Architecture and Module Interactions of the Proposed GPU Memory Sharing System

4.2 공유 객체의 생성·적재·재사용 및 생명주기

GPU 작업 프로세스는 공유 키, 데이터 크기 및 GPU 장치 번호를 중앙 관리 프로세스에 전달한다. 중앙 관리 프로세스는 동일한 키의 객체가 없으면 객체를 생성하고 최초 요청 프로세스에 최초 적재 프로세스(CREATOR) 역할을 부여한다. 동일한 키와 크기의 객체가 존재하면 후속 요청 프로세스에 재사용 프로세스(REUSER) 역할을 부여하며, 크기가 다르면 SIZE_MISMATCH로 거부한다. 이 판정은 키와 크기의 일치만 확인하며 데이터 내용의 해시·버전은 검증하지 않는다. 따라서 같은 키·크기로 서로 다른 내용을 요청하면 이를 구분하지 못하므로, 호출자는 동일한 내용에만 같은 키를 사용하고 내용이 변경되면 별도의 키로 등록해야 한다. 두 역할은 작업 프로세스에 고정되지 않고 공유 객체별로 결정된다. 동일 키의 조회·신규 객체 등록은 객체 테이블 잠금으로 보호하며, 최초 등록 세션을 CREATOR로 기록한다. 객체별 잠금으로 상태와 세션 참조를 갱신하고, 생성 중인 객체에 대한 후속 요청은 조건 변수로 생성 상태 전환을 기다린다.

클라이언트 라이브러리와 중앙 관리 프로세스는 유닉스 도메인의 SOCK_SEQPACKET 소켓으로 객체 할당, 데이터 적재 및 해제 요청을 교환한다. 새로운 객체를 생성할 때 중앙 관리 프로세스는 cudaMalloc()으로 원본 GPU 메모리를 할당하고 CUDA IPC 핸들을 생성한다. 최초 적재 프로세스가 공유 기본 포인터를 대상으로 객체 전체 크기의 H2D cudaMemcpy()를 호출하면, 중재 모듈은 이를 최초 적재 경로로 재지정한다. 클라이언트 라이브러리는 memfd_create()로 생성한 매핑 영역에 데이터를 복사하고 SCM_RIGHTS로 파일 디스크립터를 전달하며, 중앙 관리 프로세스는 이를 매핑하여 실제 H2D 전송을 수행한다[14,15]. 전송과 동기화가 성공하면 객체를 사용 중(IN_USE) 상태로 전환하고 상태 변경을 알린다.

재사용 프로세스는 기존 CUDA IPC 핸들로 공유 객체에 연결하며, 중앙 관리 프로세스는 재사용 요청 처리 시 조건 변수로 적재 완료를 기다린다. 사용 가능 상태에서는 중간 복사와 추가 H2D 전송을 생략하고, 객체 오류·해제 또는 적재 대기시간 초과 시에는 오류를 반환한다. 참조 수는 해당 객체에 등록된 세션 참조의 개수로 관리한다. 정상 해제 시 작업 프로세스는 GPU 작업을 동기화하고 로컬 CUDA IPC 매핑을 닫은 뒤 RELEASE를 전송한다. 중앙 관리 프로세스는 해당 세션 참조를 제거하며, 다른 세션의 참조가 남으면 원본 GPU 메모리를 유지하고 마지막 참조가 해제되면 회수한다. 현재 구현은 연결 종료 시 확인한 잔여 세션 참조를 경고로 기록하지만 자동으로 제거하지 않으므로, RELEASE 없이 비정

www.kci.go.kr

상 종료하면 관리자 실행 중 해당 참조와 원본 GPU 메모리가 남을 수 있다. 적재 대기시간 제한과 오류 반환은 비정상 종료 시 자동 참조 회수나 복구를 보장하지 않는다. Algorithm 1은 정상 요청 처리와 명시적 RELEASE에 따른 생명주기를 요약한다.

Algorithm 1: Shared Object Creation, Loading, Reuse, and Release	
1	UseSharedObject(request) {
2	object = FindObject(request.key);
3	if (SizeMismatch(object, request))
4	return SIZE_MISMATCH;
5	(object, role) = ResolveObject(object, request);
6	RegisterReference(object, request.session);
7	shared = OpenMapping(object.ipcHandle);
8	if (role == CREATOR)
9	LoadAndPublish(object, request.data);
10	else
11	WaitUntilInUse(object);
12	ExecuteWithSharedData(shared);
13	SynchronizeAndClose(shared);
14	SendReleaseRequest(object);
15	}
16	HandleRelease(object, session) {
17	RemoveReference(object, session);
18	if (NoReferences(object))
19	FreeSharedObject(object);
20	}

4.3 공유 포인터 인지형 API 중재

중재 모듈은 클라이언트 라이브러리에 등록된 공유 포인터 정보를 이용하여 cudaMemcpy()의 목적지가 공유 객체에 포함되는지 확인한다. 목적지가 공유 객체가 아니면, 공유 객체를 원본으로 하는 D2H 읽기를 포함하여 호출을 CUDA Runtime으로 전달한다. 목적지가 공유 객체의 정확한 기본 포인터이고, 객체 전체 크기의 H2D 전송이며, 원본이 비공유 호스트 포인터인 경우에만 요청 프로세스의 역할과 객체 상태에 따라 최초 적재 또는 재사용 경로로 재지정한다. 공유 객체를 목적지로 하는 cudaMemcpy() 중 위 조건을 만족하지 않는 호출은 거부한다. 이 거부 규칙은 중재 대상 API 호출에 한정되며, GPU 커널 내부의 직접 쓰기는 차단하지 않으므로 공유 데이터의 읽기 전용 사용은 응용이 준수해야 한다. cudaFree()는 일반 GPU 포인터이면 기존 Runtime으로 전달하고, 공유 기본 포인터이면 참조 해제 경로로 재지정하며, 공유 객체 내부 주소이면 거부한다.

적용에는 공유 객체의 명시적 요청·등록, 동적 CUDA Runtime 연결 및 LD_PRELOAD 설정이 필요하다. 등록된 객체에 대한 기존 cudaMemcpy()·cudaFree() 호출 형식은 유지하지만, cudaMemcpyAsync()와 스트림 별 처리는 중재 대상에 포함하지 않으며 실제 AI 프레임워크 연동은 검증하지 않았다.

5. 실험

5.1 실험 환경과 비교 방식

실험은 단일 GPU 서버를 논리적 엣지 노드로 간주한 통제 환경에서 수행하였으며, 하드웨어와 소프트웨어 환경은 〈Table 2〉과 같다. 작업 프로세스는 동적 CUDA Runtime인 libcudart.so.12에 연결하였고, 제안 방식에서는 LD_PRELOAD로 선택적 API 중재 모듈을 적재하였다. GPU 상태는 NVML을 이용하여 50~100 ms 간격으로 수집하였다.

〈Table 2〉 Experimental Environment

Category	Configuration
CPU	AMD Ryzen 9 9950X, 16 cores
System Memory	64 GiB
GPU	NVIDIA GeForce RTX 4090, 24,564 MiB
Operating System	Ubuntu 24.04.4 LTS
CUDA Toolkit	CUDA 12.0

독립 적재 방식은 각 작업 프로세스가 공용 GPU 메모리를 개별 할당하고 동일한 데이터를 H2D 전송하였다. 제안 방식은 중앙 관리 프로세스가 공용 데이터를 한 번 적재하고 작업 프로세스가 CUDA IPC로 기존 공유 객체에 연결하였다. 두 방식은 동일한 실행 파일, 입력 데이터, GPU 커널 및 검증 방식을 사용하였으며 공용 데이터의 할당·준비 경로만 달리하였다.

워크로드는 32비트 부동소수점 행렬을 공용 데이터로 사용하는 GEMV(General Matrix-Vector Multiplication) 연산 y = Wx로 구성하였다. GEMV는 동일한 공용 행렬을 사용하는 독립 프로세스에서 입력 데이터와 GPU 커널을 유지하면서 공용 데이터의 할당·준비 경로만 변경할 수 있으므로, 중복 GPU 메모리 할당과 반복 H2D 전송 감소 효과를 분리하여 평가하기 위한 통제 워크로드로 선정하였다. 따라서 본 실험은 공용 데이터 공유 기법의 효과를 검증하기 위한 것이며, 실제 AI 모델의 전체 추론 성능이나 프레임워크 연동성을 평가한 것은 아니다. 메모리·용량·커널 실험은 행 수를 4,096, 재사용 실험은 1,024로 고정하고, 열 수를 조절하여 256 MiB~4 GiB의 공용 행렬을 구성하였다. 행렬과 입력 벡터는 결

www.kci.go.kr

정적으로 생성하였으며, 커널은 행별 1개 block과 block당 256개 thread를 사용하였다. 계산값의 절대오차가 max(0.05, |기준값| × 10^{-3})을 초과하면 불일치로 판정하였고, 재사용 실험의 표본 D2H(Device-to-Host) 검증에는 절대오차 10^{-6}을 적용하였다. 이는 연산·전송 결과의 정확성 확인이며, 객체 재사용 승인 시 데이터 내용의 동일성을 자동으로 판정하는 기능은 아니다.

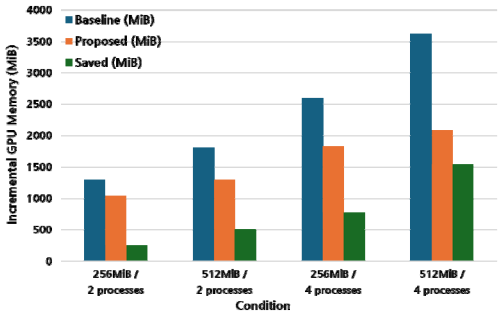
증분 GPU 메모리는 READY_HOLD 구간의 중앙값에서 IDLE 구간의 중앙값을 뺀 값이며, 총 GPU 메모리는 READY_HOLD 구간의 중앙값이다. 총 GPU 메모리에는 배경 사용량과 작업 프로세스의 CUDA 실행 문맥·전용 버퍼가 포함되고, 제안 방식에는 중앙 관리 프로세스의 CUDA 실행 문맥도 포함된다.

재사용 실험에서는 참조 유지 프로세스가 최초 적재 후 공유 객체를 유지하였다. 각 반복에서 측정 제외 요청 3회 후 20개 요청을 측정하고 그 평균을 하나의 표본으로 사용하여 동일 반복 번호의 두 방식을 대응시켰다. 감소율은 95% 신뢰구간과 정확 Wilcoxon 부호순위검정으로 분석하였다. 장시간 커널 실험은 준비 실행 10회 후 1,500회의 누적 커널 시간을 CUDA Event로 측정하였다. 자원 회수는 실험 종료 후 CUDA 연산 프로세스 수가 0이고 GPU 메모리가 IDLE 중앙값 + 64 MiB 이하로 복귀한 상태가 연속 세 표본에서 유지되는지 확인하였다. 이 검증은 종료 후의 자원 복귀를 확인한 것으로, 관리자 실행 중 비정상 종료 세션의 자동 참조 회수나 장애 복구를 검증한 것은 아니다.

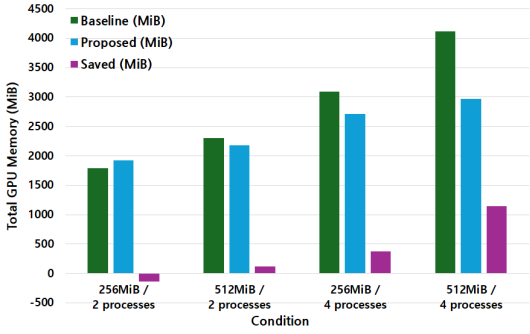
5.2 GPU 메모리 사용량과 동시 실행 용량

GPU 메모리 사용량 실험은 256 MiB와 512 MiB 공용 데이터에 대해 작업 프로세스 2개와 4개 조건을 각각 3회 수행하였다. 공용 데이터의 논리적 GPU 메모리 사용량은 작업 프로세스 2개와 4개에서 각각 50%와 75% 감소하였다. [Fig. 2](a)와 같이 증분 GPU 메모리는 시험한 네 조건에서 19.89~42.46% 감소하였으며, 512 MiB 데이터를 4개 프로세스가 사용한 조건에서는 3,624.19 MiB에서 2,085.31 MiB로 감소하였다.

[Fig. 2](b)는 중앙 관리 프로세스의 CUDA 실행 문맥을 포함한 총 GPU 메모리를 나타낸다. 제안 방식은 약 394 MiB의 고정 GPU 메모리를 추가로 사용하므로 256 MiB×2개 프로세스 조건에서는 총 메모리가 7.55% 증가하였다. 반면 나머지 세 조건에서는 공용 데이터 중복분의 절감량이 고정비용보다 커져 총 GPU 메모리가 5.24~27.82% 감소하였다. 따라서 논리적·증분 메모리는 모든 조건에서 감소하지만, 총 메모리의 감소 여부는 공용 데이터 크기와 프로세스 수에 따라 달라진다.



(a) Incremental GPU Memory Usage from IDLE to READY_HOLD



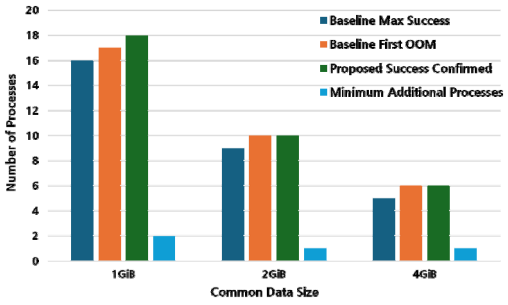
(b) Total GPU Memory Usage during READY_HOLD

[Fig. 2] GPU Memory Usage

용량 경계 실험은 1 GiB×17개·18개 프로세스, 2 GiB×10개 프로세스 및 4 GiB×6개 프로세스의 네 조건에서 각각 5쌍씩 수행하였다. 독립 적재 방식은 20회 모두 cudaErrorMemoryAllocation으로 실패했지만 제안 방식은 대응하는 20회를 모두 완료하였다. [Fig. 3]과 같이 독립 적재 방식은 1 GiB, 2 GiB 및 4 GiB에서 각각 16개, 9개 및 5개 프로세스까지 성공하였고, 제안 방식은 시험 범위에서 각각 18개, 10개 및 6개까지 성공하였다. 제안 방식의 값은 실제 최대치가 아니라 시험 범위에서 확인한 하한이다.

제안 방식의 20회 실험에는 최초 적재 프로세스 20개와 재사용 프로세스 235개가 포함되었으며, 공용 데이터의 실제 적재는 실험별 한 번씩 총 20회만 수행되었다. 제안 방식의 20회 실행에서 계산 결과 불일치와 API 재지정 실패는 발생하지 않았다. 독립 적재 방식을 포함한 전체 40회 실행은 모두 5.1절의 실험 종료 후 자원 회수 기준을 충족하였다.

www.kci.go.kr

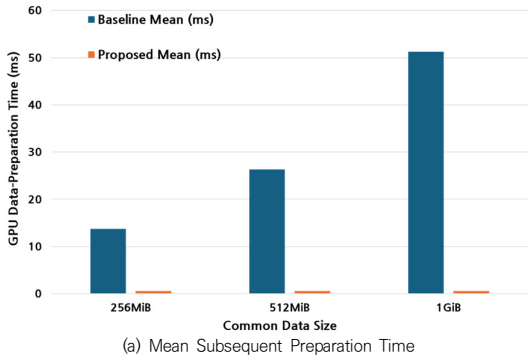


[Fig. 3] Concurrent Execution Capacity by Common-Data Size

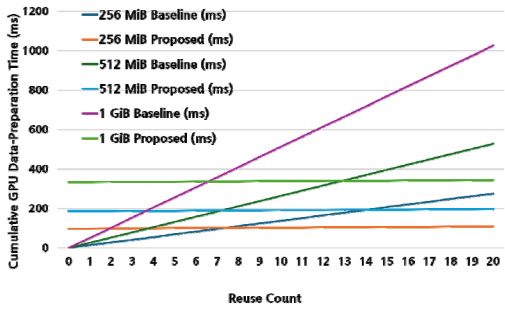
5.3 데이터 재사용과 시스템 오버헤드

공용 데이터가 이미 적재된 조건에서 독립 적재 방식의 GPU 데이터 준비 시간은 256 MiB, 512 MiB 및 1 GiB에서 각각 13.748 ms, 26.368 ms 및 51.253 ms였다. 제안 방식은 각각 0.585 ms, 0.545 ms 및 0.554 ms로 측정되어 준비 시간이 95.74%, 97.93% 및 98.92% 감소하였다. CUDA IPC 연결 시간은 독립 적재 방식의 cudaMalloc() 시간보다 0.31~0.35 ms 길었지만, 데이터 크기에 비례하는 반복 H2D 전송을 생략하여 전체 준비 시간을 줄였다. 10개 대응 실험에서 제안 방식의 준비 시간이 모두 짧았으며, 감소율의 95% 신뢰구간은 256 MiB, 512 MiB 및 1 GiB에서 각각 95.624~95.860%, 97.859~98.008% 및 98.896~98.943%였다. 정확 Wilcoxon 부호순위검정의 유의확률은 세 조건 모두 0.001953이었다. 평균 준비 시간은 [Fig. 4](a)에 나타낸다.

최초 적재 비용과 후속 요청 20회를 포함한 누적 준비 시간은 독립 적재 방식보다 256 MiB, 512 MiB 및 1 GiB에서 각각 60.21%, 62.66% 및 66.40% 짧았다. [Fig. 4](b)와 같이 제안 방식의 누적 준비 시간이 독립 적재 방식보다 처음 짧아지는 재사용 횟수는 256 MiB에서 8회, 512 MiB에서 7~8회, 1 GiB에서 7회였다. 이 결과는 최초 GPU 데이터 적재와 후속 GPU 데이터 준비 경로만을 비교한 것이며, 중앙 관리 프로세스와 참조 유지 프로세스의 전체 시작·유지 비용을 포함한 서비스 수준의 손익분기점은 아니다. 제안 방식의 누적 준비 시간은 최초 공유 객체 적재 비용과 후속 R회 요청의 준비 시간 합으로 정의하였다. 독립 적재 방식은 대응하는 R회 요청에서 발생한 개별 데이터 준비 시간의 합으로 정의하였다.



(a) Mean Subsequent Preparation Time



(b) Cumulative Preparation Time Including the Initial Loading Cost

[Fig. 4] GPU Data-Preparation Performance

장시간 GEMV 실험은 512 MiB 공용 데이터와 작업 프로세스 4개 조건에서 각 방식을 3회 반복하였다. 제안 방식의 커널 시간과 GFLOPS 변화는 독립 적재 방식 대비 각각 −0.033%와 +0.033%로 미미하였다. 평균 GPU 활용률은 독립 적재 방식 95.50%, 제안 방식 88.64%였으므로 GPU 연산 활용률 향상을 결과로 해석하지 않았다. 시험한 기능·성능 실험에서는 계산 결과 불일치와 API 재지정 실패가 발생하지 않았다. 본 실험의 주요 효과는 커널 속도 향상이 아니라 공용 데이터의 중복 GPU 메모리 할당과 반복 H2D 전송의 감소이며, 실제 AI 응용의 추론 성능에 대한 효과는 별도 검증이 필요하다.

6. 결론

본 연구는 단일 GPU에서 여러 프로세스가 동일한 대규모 읽기 전용 데이터를 반복적으로 적재할 때 발생하는 GPU 메모리 중복과 H2D 전송을 줄이기 위해 중앙 관리 프로세스, CUDA IPC 및 선택적 API 중재를 결합한 데이터 공유 기법을 제안하였다. 중앙 관리 프로세스는 원본 GPU 메모리와 공유 객체의 참조 생명주기를 관

www.kci.go.kr

리하고, 최초 적재 프로세스의 유효한 H2D 호출은 실제 데이터 전송으로 처리하며 재사용 프로세스의 동일 호출은 기존 객체의 사용 가능 상태를 확인하는 경로로 전환한다. 공용 데이터의 논리적 GPU 메모리 사용량은 작업 프로세스 2개와 4개에서 각각 50%와 75% 감소하였다. 중앙 관리 프로세스의 약 394 MiB 고정 GPU 메모리로 인해 256 MiB×2개 프로세스 조건에서는 총 GPU 메모리가 7.55% 증가하였으나, 나머지 세 조건에서는 5.24~27.82% 감소하였다. 네 개의 용량 경계 조건에서 수행한 20쌍의 비교에서 독립 적재 방식은 모두 GPU 메모리 부족으로 실패한 반면 제안 방식은 모두 정상적으로 완료하였다. 두 방식을 합한 전체 40회 실행은 5.1절에서 정의한 실험 종료 후 자원 회수 기준을 충족하였다. 후속 GPU 데이터 준비 시간은 95.74~98.92% 감소하였으며, 누적 준비 시간은 GPU 데이터 준비 경로에서 7~8회 재사용 이후 독립 적재 방식보다 짧아졌다. 장시간 커널 성능 차이는 미미하였고 계산 결과 불일치와 API 재지정 실패는 발생하지 않았다.

본 결과는 동일 호스트의 단일 GPU에서 연속된 읽기 전용 공용 데이터를 반복 사용하는 통제형 GEMV 환경에 한정된다. 현재 구현은 같은 공유 키와 데이터 크기가 동일한 내용을 나타내고, 응용이 공유 데이터를 읽기 전용으로 사용한다는 전제에 의존한다. 데이터 내용의 해시·버전 검증과 GPU 커널 내부의 직접 쓰기 차단은 제공하지 않는다. 또한 RELEASE 없이 비정상 종료한 세션의 참조를 자동 회수하지 않으므로 관리자 실행 중 원본 GPU 메모리가 남을 수 있다. 비동기 전송과 스트림별 호출에 대한 중재 및 중앙 관리 프로세스 재시작은 지원하지 않으며, 실제 AI 모델·프레임워크, 컨테이너 및 다중 GPU 환경은 평가하지 않았다.

향후에는 실제 호스트 데이터의 해시값을 비교하여 동일한 키·크기로 요청된 서로 다른 데이터를 검출하는 재사용 검증 기법을 연구할 필요가 있다. 또한 CREATOR·REUSER의 비정상 종료 시 프로세스 종료와 공유 메모리의 사용 종료를 구분하여 확인하고, 다른 정상 사용자의 접근을 보존하면서 잔여 세션 참조를 회수하는 방안을 검토할 필요가 있다. 이때 내용 검증에 따른 추가 준비 비용과 장애 주입 실험을 통한 자원 회수·복구의 안전성을 평가해야 한다.

아울러 기존 AI 프레임워크 기반 모델에 적용하기 위해 가중치 메모리의 등록·적재·해제와 실행·동기화 조건을 제안 시스템의 공유 객체 관리와 연동하는 확장 연구를 수행할 계획이다. 이를 바탕으로 동일 가중치를 사용하는 다중 추론 환경에서 GPU 메모리 사용량, 모델 준비 시간 및 추론 성능을 평가할 예정이다.

REFERENCES

[1] NVIDIA Corporation, "CUDA C++ Programming Guide," NVIDIA Developer Documentation, accessed Aug. 2026.

[2] NVIDIA Corporation, "CUDA Runtime API," NVIDIA Developer Documentation, accessed Aug. 2026.

[3] X.Z.Liu, Y.H.Zhao, S.F.Liu, X.Li, Y.B.Zhu, X.Liu and X.Jin, "MuxFlow: Efficient GPU Sharing in Production-Level Clusters with More Than 10000 GPUs," Science China Information Sciences, Vol.67, Article No.222101, 2024.

[4] Z.Y.Han, R.T.Zhou, C.Z.Xu, Y.F.Zeng and R.L.Zhang, "InSS: An Intelligent Scheduling Orchestrator for Multi-GPU Inference With Spatio-Temporal Sharing," IEEE Transactions on Parallel and Distributed Systems, Vol.35, No.10, pp.1735-1748, 2024.

[5] Y.Q.Peng, W.G.Gao and H.C.Peng, "Serving DNN Inference With Fine-Grained Spatio-Temporal Sharing of GPU Servers," IEEE Transactions on Services Computing, Vol.17, No.6, pp.4310-4323, 2024.

[6] S.Y.Woo, J.H.Yeo, J.H.Kim and K.W.Lee, "Exploring GPU Sharing Techniques for Edge AI Smart City Applications," ETRI Journal, Vol.47, No.5, pp.855-864, 2025.

[7] H.R.Gao, D.H.Luo, Y.W.Wu, H.Wu and W.B.Zhang, "vTensor-Based GPU Memory Management for Edge Deep Learning Training," Journal of Computer Science and Technology, Vol.40, No.6, pp.1608-1625, 2025.

[8] C.Y.Shin, Y.H.Go, Y.H.Yoo, J.W.Jeong, J.H.Hwang, G.S.Yang and C.Yoo, "Prediction-Based GPU Sharing for Distributed Training," Future Generation Computer Systems, Vol.181, Article No.108413, 2026.

[9] S.C.Chen, D.S.Wang, W.Z.Zhang, M.Hao, Y.C.Tian and H.He, "Performance Prediction of Concurrent DNN Training Tasks in GPU Spatial Sharing Environments," ACM Transactions on Architecture and Code Optimization, Vol.23, No.2, Article No.59, 2026.

[10] J.Villarrubia, L.Costero, F.D.Igual and K.Olcoz, "A Comprehensive Evaluation of Spatial Co-Execution on GPUs Using MPS and MIG Technologies," The Journal of Supercomputing, Vol.82, Article No.175, 2026.

[11] NVIDIA Corporation, "NVIDIA Management Library (NVML) API Reference Guide," NVIDIA Developer Documentation, accessed Aug. 2026.

[12] The Linux man-pages Project, "ld.so(8): Dynamic Linker/Loader," Linux man-pages, accessed Aug. 2026.

www.kci.go.kr

[13] The Linux man-pages Project, "dlsym(3): Obtain Address of a Symbol in a Shared Object or Executable," Linux man-pages, accessed Aug. 2026.

[14] The Linux man-pages Project, "memfd_create(2): Create an Anonymous File," Linux man-pages, accessed Aug. 2026.

[15] The Linux man-pages Project, "unix(7): Sockets for Local Interprocess Communication," Linux man-pages, accessed Aug. 2026.

강 지 훈(Jihun Kang) [정회원]



■ 2013년 2월 : 배재대학교 게임멀티미디어공학과 (공학석사)
■ 2020년 고려대학교 컴퓨터학과 (공학박사)
■ 2020년 8월 ~ 2024년 2월 : 고려대학교 4단계 BK21컴퓨터학교육연구단 연구교수
■ 2024년 9월 ~ 현재 : 한국방송통신대학교 컴퓨터과학과 교수

〈관심분야〉
클라우드 시스템, 고성능 컴퓨팅, GPU 가상화

www.kci.go.kr