논문 2026-2-13 http://dx.doi.org/10.29056/jsf.2026.06.13

GPU 메모리 아키텍처 상이성에 따른 대규모 격자 처리의 이식 성능 분석

김민수*, 김재웅*† , 장미영*

Analysis of Portability Performance of Large-Scale Grid Processing Based on GPU Memory Architecture Differences

Min-Soo Kim*, Jae-Woong Kim*† , Mi-young Jang*

요 약

본 논문은 PC 이산형 GPU 환경에서 작성된 대규모 격자 재표본화 CUDA 코드를 Jetson AGX Orin 기반 UMA GPU 환경으로 이식할 때, 메모리 방식과 접근 패턴에 따른 성능 특성을 분석하였다. PC는 CPU 메모리와 GPU 전용 메모리가 분리되어 있는 반면, Jetson은 CPU와 GPU가 물리 메모리를 공유하므로 동일한 메모리 방식이 두 환경에서 같은 결과를 보장하지 않는다. 이를 위해 명시적 복사 방식(Device Memory), 통합 메모리 방식(Unified Memory), 제로 카피 방식(Zero-copy)을 적용하고, 연속 접근(linear), 국소 비연속 접근(local-offset), 분산 접근(scatter) 조건에서 실행시간과 메모리 접근 특성을 비교하였다. 실험 결과 PC 환경에서는 명시적 복사 방식이 가장 안정적인 성능을 보였으며, Jetson 환경에서는 대용량 조건과 메모리 운용 방식에 따라 통합 메모리 및 제로 카피 방식의 활용 가능성이 확인되었다. 분산 접근에서는 두 환경 모두 성능 저하가 나타났으며, CUDA 코드 이식 시 메모리 구조와 함께 데이터 접근 패턴을 고려할 필요가 있음을 확인하였다.

Abstract

This paper analyses the performance characteristics observed when porting large-scale grid-resampling CUDA code from a PC discrete-GPU environment to a Jetson AGX Orin-based UMA GPU environment. In a PC environment, CPU memory and dedicated GPU memory are physically separated, whereas the Jetson shares physical memory between the CPU and GPU. Therefore, the same memory method does not necessarily produce equivalent performance across the two platforms. Explicit-copy memory (Device Memory), Unified Memory, and Zero-copy were evaluated under linear, local-offset, and scatter access patterns. The results showed that Device Memory provided the most stable performance on the PC. On the Jetson, Unified Memory and Zero-copy showed potential advantages depending on data size and memory-management conditions. Scatter access degraded performance on both platforms, indicating that both memory architecture and data-access patterns should be considered when porting CUDA code.

한글키워드 : GPU 메모리 아키텍처, CUDA 메모리 방식, 제로 카피, 대규모 격자 처리, 이식 성능 분석

keywords : GPU Memory Architecture, CUDA Memory Management, Zero-copy, Large-scale Grid Processing, Portability Performance Analysis

* 국립공주대학교 컴퓨터공학과
† 교신저자: 김재웅(email: jykim@kongju.ac.kr)

접수일자: 2026.06.01. 심사완료: 2026.06.15.
게재확정: 2026.06.20.

www.kci.go.kr

1. 서 론

최근 위성영상, 공간정보, 자율주행 인식 분야에서는 대규모 데이터를 빠르게 처리하는 기술의 중요성이 커지고 있다[1-4]. 이들 데이터는 픽셀 또는 격자점 단위의 좌표 변환, 보간, 재표본화 연산을 반복하므로, 해상도와 처리 범위가 커질수록 연산량과 메모리 접근량이 함께 증가한다. 이에 따라 CUDA 기반 GPU 병렬 처리는 대규모 데이터 처리를 위한 대표적 기술로 활용되고 있다[4-6].

기존 CUDA 기반 처리 코드는 주로 PC 이산형 GPU 환경을 기준으로 개발되었다. PC 환경에서는 CPU 메모리와 GPU 전용 메모리가 분리되어 있으므로, 입력 데이터를 GPU 메모리로 복사한 후 커널을 실행하고 결과를 다시 CPU 메모리로 가져오는 방식이 일반적이다[5,6]. 이 방식은 복사 비용이 발생하지만, 커널 수행 시 GPU 전용 메모리를 사용하므로 비교적 안정적인 접근 성능을 기대할 수 있다[7].

반면 Jetson과 같은 임베디드 GPU 보드는 제한된 전력과 공간에서 GPU 연산을 수행하는 엣지 컴퓨팅 플랫폼으로 활용된다[8-10]. NVIDIA Jetson AGX Orin은 Ampere 기반 GPU와 LPDDR5 공유 메모리를 사용하며[11], CPU와 GPU가 동일한 물리 메모리를 공유하는 UMA 구조를 가진다[12-14]. 따라서 PC 환경에서 사용한 CUDA 메모리 방식이 Jetson에서도 동일한 성능 특성을 보인다고 보기 어렵다.

CUDA 메모리 방식은 데이터 이동과 접근 구조에 따라 차이가 있다. 명시적 복사 방식(Device Memory)은 CPU 메모리와 GPU 메모리 사이의 이동을 직접 수행한다. 통합 메모리 방식(Unified Memory)은 코드 이식성을 높이지만 접근 주체 변경 시 관리 비용이 발생할 수 있다. 제로 카피 방식(Zero-copy)은 GPU가 호스트 메모리를 직접 참조하여 복사를 줄일 수 있으나, 플랫폼 구조와 접근 패턴에 따라 성능이 달라질 수 있다[13,14]. 또한 대규모 격자 재표본화에서는 연속 접근(linear), 국소 비연속 접근(local-offset), 분산 접근(scatter)과 같은 메모리 접근 패턴이 처리 성능에 영향을 준다.

본 논문의 차별성은 PC와 Jetson 환경을 독립적으로 비교하는 데 그치지 않고, PC 이산형 GPU 환경에서 작성된 동일 CUDA 격자 재표본화 코드를 Jetson AGX Orin 기반 UMA GPU 환경으로 이식하는 상황을 대상으로 한 데 있다. 이를 위해 명시적 복사 방식, 통합 메모리 방식, 제로 카피 방식을 동일한 CPU-GPU 교차 처리 흐름에서 비교하고, 접근 패턴별 실행시간과 메모리 효율 변화를 함께 분석하였다.

따라서 본 연구는 PC와 Jetson의 단순 성능 차이를 확인하는 것이 아니라, 동일한 대규모 격자 처리 코드가 서로 다른 메모리 아키텍처에서 실행될 때 메모리 운용 방식과 접근 패턴에 따라 성능 특성이 어떻게 달라지는지를 분석한다. 이를 통해 기존 PC 환경의 CUDA 코드를 임베디드 UMA GPU 환경으로 이식할 때, 메모리 구조와 접근 패턴을 함께 고려해야 함을 확인하고자 한다.

2. 관련 연구

2.1 GPU 메모리 아키텍처 연구

GPU 메모리 아키텍처는 CUDA 코드의 실행 성능에 직접적인 영향을 준다. PC 환경의 이산형 GPU는 CPU 메모리와 GPU 전용 메모리가 물리적으로 분리되어 있어, 데이터 처리 시 H2D (Host-to-Device) 및 D2H(Device-to-Host) 복사가 필요하다[5,6]. GPU 전용 메모리는 높은 대역폭을 제공하지만, CPU와 GPU 사이의 데이터 이동이 반복되면 복사 비용이 전체 실행시간에 영

www.kci.go.kr

향을 줄 수 있다.

최근 GPU는 L1/L2 캐시와 메모리 계층을 개선하여 데이터 접근 성능을 높이고 있다[15]. Hopper GPU를 분석한 최근 연구에서도 메모리 계층, L2 캐시, 전역 메모리 접근 특성이 GPU 세대와 접근 방식에 따라 달라짐을 보였다[19]. 그러나 이산형 GPU는 CPU 메모리와 GPU 메모리가 분리된 구조를 유지하므로, 통합 메모리 방식(Unified Memory)에서도 페이지 이동(page migration)과 접근 주체 전환 비용이 발생할 수 있다[12,14].

반면 임베디드 GPU 플랫폼은 제한된 전력과 공간에서 GPU 연산을 수행하기 위한 엣지 컴퓨팅 장치로 활용되고 있다[8,9,10,16]. Jetson AGX Orin과 같은 플랫폼은 CPU와 GPU가 동일한 LPDDR5 메모리를 공유하는 UMA 구조를 사용한다[11,13]. 이 구조는 명시적 복사를 줄일 수 있으나, CPU와 GPU가 메모리 대역폭을 공유하므로 접근 패턴과 데이터 사용 순서에 따라 성능 차이가 발생할 수 있다. 따라서 동일한 CUDA 코드와 메모리 방식을 사용하더라도 이산형 GPU와 UMA GPU에서는 서로 다른 성능 특성이 나타날 수 있다. 그림 1에서는 PC 이산형 GPU와 Jetson UMA GPU의 물리적 메모리 구성을 비교하였다.

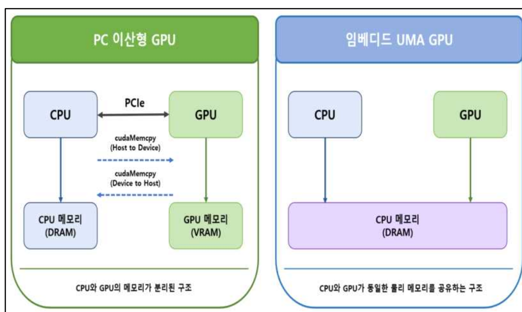

그림 1. 물리적 메모리 아키텍처 비교
Fig. 1. Physical Memory Architecture Comparison

2.2 CUDA 메모리 관리 방식 연구

CUDA 메모리 관리 방식은 명시적 복사 방식(Device Memory), 통합 메모리 방식(Unified Memory), 제로 카피 방식(Zero-copy)으로 구분할 수 있으며, 각 방식은 데이터 배치와 CPU-GPU 접근 구조에서 차이를 가진다[6]. 표 1은 CUDA 메모리 방식별 특징을 나타낸 것이다.

표 1. CUDA 메모리 방식별 특징
Table 1. Characteristics of CUDA Memory Management Methods

방식	주요 함수	기존 연구에서의 특징
명시적 복사 방식 (Device Memory)	cudaMalloc, cudaMemcpy	명시적 복사 비용 존재, GPU 메모리 접근 안정적
통합 메모리 방식 (Unified Memory)	cudaMallocManaged	코드 이식성 우수, 페이지 관리 비용 발생 가능
제로 카피 방식 (Zero-copy)	cudaHostAlloc, cudaHostGetDevicePointer	복사 감소 가능, 직접 접근 지연 발생 가능

명시적 복사 방식은 호스트 메모리와 GPU 메모리를 분리하고 H2D 및 D2H 복사를 수행한다[5,6]. 통합 메모리 방식은 CPU와 GPU가 동일한 주소 공간을 사용하지만, 접근 주체가 변경될 때 페이지 이동과 동기화 비용이 발생할 수 있다[12,14]. 제로 카피 방식은 GPU가 고정된 호스트 메모리를 직접 참조하므로, 명시적 복사는 줄일 수 있으나 이산형 GPU에서는 PCIe 기반 접근 지연이 커질 수 있다[6,13,14]. 최근 HMM (Heterogeneous Memory Management) 기반 공

www.kci.go.kr

유 가상 메모리 연구에서는 페이지 오류, 페이지 이동, 퇴출 정책이 접근 패턴과 데이터 재사용 특성에 따라 성능 병목을 유발할 수 있음을 분석하였다[20]. 이는 CUDA 통합 메모리 방식과 구현 구조는 다르지만, CPU-GPU 간 데이터 이동과 접근 전환 비용을 접근 패턴과 함께 해석해야 한다는 점에서 본 연구와 관련된다. 따라서 메모리 방식은 복사 여부만으로 판단하기보다 플랫폼 구조와 접근 패턴을 함께 고려할 필요가 있다.

2.3 대규모 격자 처리 연구

대규모 격자 재표본화는 위성영상 정사보정, 기하보정, 좌표변환 과정에서 반복적으로 사용된다. 출력 격자점마다 원본 영상의 참조 좌표를 계산하고, 최근접 또는 양선형 보간을 위해 인접 화소를 읽으므로 대량의 좌표 계산과 메모리 접근이 반복된다. SAR 영상과 같은 고해상도 원격탐사 자료는 처리 대상 화소 수가 많아 GPU 가속이 효과적으로 활용될 수 있다[4].

재표본화 과정의 성능은 연산량뿐 아니라 접근 패턴에 영향을 받는다. 연속 접근(linear)은 인접 스레드가 연속된 주소를 참조하여 메모리 병합 접근에 유리하다. 국소 비연속 접근(local-offset)은 보간 과정에서 주변 화소를 참조하는 상황을 단순화한 패턴이다. 분산 접근(scatter)은 좌표변환 결과에 따라 입력 참조 위치가 넓게 분산되는 경우를 모사하며, 캐시 효율과 메모리 병합 효율을 낮춰 커널 성능 저하를 유발할 수 있다[7].

기존 연구는 GPU 메모리 구조, CUDA 메모리 방식, 대규모 격자 처리의 병렬화 가능성을 각각 다루어 왔다. 그러나 동일한 격자 재표본화 워크로드를 이산형 GPU와 UMA GPU에서 실행하고, 메모리 방식 및 접근 패턴에 따른 데이터 이동, 커널 실행, CPU-GPU 접근 전환 비용을 함께 비교한 연구는 제한적이다. 본 연구는 이러한 차이를 동일 코드와 동일 접근 패턴 조건에서 분석한다.

3. 실험 구성 및 측정 방법

그림 2는 검증용 CUDA 코드의 데이터 흐름을 나타낸다. 본 연구는 PC 이산형 GPU와 Jetson AGX Orin 기반 UMA 환경에서 동일한 격자 재표본화 코드를 실행하여, 메모리 구조와 접근 방식에 따른 성능 변화를 분석하였다. 플랫폼 간 절대 성능 비교보다 CPU-GPU 간 데이터 이동, 접근 주체 전환, 커널 실행 구간의 차이를 확인하는 데 목적을 두었다.

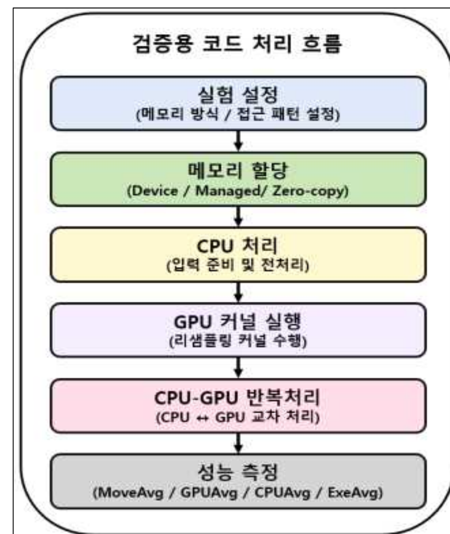

그림 2. 검증용 CUDA 코드의 처리 흐름
Fig. 2. Workflow of the CUDA Validation Code

3.1 실험 플랫폼

표 2는 실험 플랫폼 사양을 나타낸다. 본 실험에서는 PC 환경으로 GeForce RTX 4060을 사용하였고, Jetson 환경은 NVIDIA Jetson AGX Orin 32GB 모델을 기준으로 하였다. Jetson

www.kci.go.kr

AGX Orin은 Ampere 기반 GPU와 LPDDR5 공유 메모리를 사용하는 임베디드 GPU 플랫폼이다[11].

표 2. 실험 플랫폼 사양
Table 2. Experimental Platform Specifications

구분	PC 환경	Jetson 환경
GPU	GeForce RTX 4060	Jetson AGX Orin
CUDA Core	3,072개	1,792개
메모리	8GB GDDR6 VRAM	32GB LPDDR5 공유 메모리
대역폭	272.0 GB/s	204.8 GB/s
메모리 구조	CPU DRAM-GPU VRAM 분리	CPU-GPU 공유 UMA
데이터 이동	PCIe 기반 명시적 복사	공유 메모리 기반 접근
CUDA CC / Version	8.9 / 12.6	8.7 / 12.6

3.2 검증용 CUDA 코드 구성

본 코드는 동일한 격자 재표본화 커널(RemapKernel)에 세 가지 CUDA 메모리 방식을 적용하여 구현하였다. 비교 대상은 명시적 복사 방식 (Device Memory), 통합 메모리 방식(Unified Memory), 제로 카피 방식(Zero-copy)이며, 세 방식은 동일한 연산을 수행하고 메모리 할당 및 CPU-GPU 데이터 접근 방식만 다르다. 명시적 복사 방식(Device Memory)은 cudaMalloc()으로 GPU 메모리를 할당한 뒤, cudaMemcpy()를 이용하여 호스트 메모리와 장치 메모리 사이에서 H2D(Host-to-Device) 및 D2H(Device-to-Host) 복사를 수행한다. 연산은 GPU 메모리에서 이루어지며, CPU와 GPU가 서로 다른 메모리 공간을 사용한다.

통합 메모리 방식(Unified Memory)은 cudaMallocManaged()를 이용하여 CPU와 GPU가 동일한 주소 공간을 공유하도록 구성하였다. CPU와 GPU가 하나의 포인터를 사용하지만, 접근 주체가 변경될 경우 페이지 단위의 page migration이 발생할 수 있으며, 필요에 따라 cudaMemPrefetchAsync()를 이용하여 데이터 이동을 수행한다.

제로 카피 방식(Zero-copy)은 cudaHostAlloc()으로 고정(pinned) 호스트 메모리를 할당한 뒤, cudaHostGetDevicePointer()를 통해 GPU가 해당 메모리를 직접 참조하도록 구성하였다. 별도의 H2D 및 D2H 복사 과정 없이 CPU와 GPU가 동일한 호스트 메모리를 공유하지만, GPU는 시스템 메모리에 직접 접근하게 된다.

본 워크로드는 실제 위성영상 처리의 정사보정, 기하보정, 좌표변환 후 재표본화 단계와 연결된다. 해당 단계에서는 출력 격자점마다 원본 영상의 참조 좌표를 계산하고, 최근접 또는 양선형 보간을 위해 인접 화소를 읽는다. 따라서 연속 접근(linear), 국소 비연속 접근(local-offset), 분산 접근(scatter)은 위성영상 재표본화 과정에서 나타나는 대표적 메모리 접근 특성을 단순화한 조건이다. 다만 본 결과는 전체 위성영상 처리 체인의 실행시간을 직접 예측하는 것이 아니라, 재표본화 핵심 구간의 메모리 접근 특성을 분석한 결과로 한정한다.

그림 3은 각 메모리 방식에서 CPU와 GPU 사이의 데이터 흐름을 나타낸다. 명시적 복사 방식(Device Memory)은 CPU 메모리와 GPU 메모리가 분리되어 있으며, 입력 데이터를 GPU 메모리로 복사한 후 커널을 수행하고 결과를 다시 CPU 메모리로 복사하는 구조를 가진다. 통합 메모리 방식(Unified Memory)은 CPU와 GPU가 동일한 주소 공간을 공유하며, 접근 주체가 변경될 때 CUDA 런타임이 필요한 데이터를 자동으로 관리한다. 이 과정에서 페이지 단위의 데이터 마이그

www.kci.go.kr

레이션(page migration)이 발생할 수 있으며, CPU와 GPU가 서로 다른 위치의 데이터를 반복적으로 접근하는 경우 추가적인 관리 비용이 발생할 수 있다.

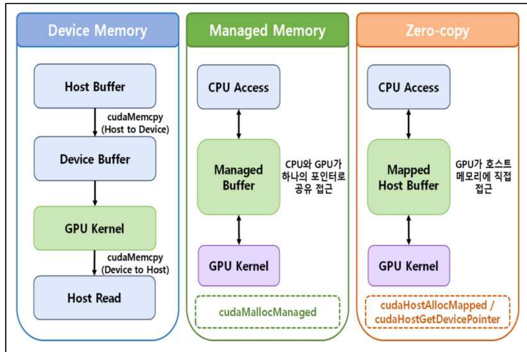

그림 3. 메모리 방식별 실행 구조

Fig. 3. Execution Structure by Memory Method

제로 카피 방식(Zero-copy)은 고정된 호스트 메모리를 GPU가 직접 참조하는 구조로, 별도의 명시적 복사 없이 CPU와 GPU가 동일한 데이터를 접근할 수 있다. 따라서 데이터 복사 비용은 줄일 수 있으나, 실제 성능은 메모리 접근 패턴과 플랫폼의 메모리 구조에 영향을 받는다.

그림 4는 메모리 방식별 핵심 코드를 나타낸 것이며, 세 가지 메모리 방식(Device Memory, Unified Memory, Zero-copy)은 모두 동일한 RemapKernel을 사용하여 격자 처리 연산을 수행하도록 구성하였다. 각 방식의 차이는 메모리 할당 방식과 CPU-GPU 간 데이터 접근 방식에만 두었으며, 이를 통해 동일한 연산 조건에서 메모리 방식에 따른 성능 차이를 비교할 수 있도록 하였다.

3.3 실험 조건

표 3은 본 실험에 적용한 주요 조건을 나타낸 것이다. 접근 패턴은 연속 접근(linear), 국소 비연속 접근(local-offset), 분산 접근(scatter)으로 구분하였다.

```
// (a) Device Memory
cudaMalloc((void**)&d_input, bytes);
cudaMalloc((void**)&d_output, bytes);

cudaMemcpy(d_input, h_input, bytes, cudaMemcpyHostToDevice);

RemapKernel << <grid, block >> > (d_input, d_output, width, height, pattern);
cudaDeviceSynchronize();

cudaMemcpy(h_output, d_output, bytes, cudaMemcpyDeviceToHost);

// (b) Managed Memory
float* input = nullptr;
float* output = nullptr;

cudaMallocManaged(&input, bytes);
cudaMallocManaged(&output, bytes);

CpuStage(input, count);

RemapKernel << <grid, block >> > (input, output, width, height, pattern);
cudaDeviceSynchronize();

CpuStage(output, count);

// (c) Zero-copy
cudaHostAlloc((void**)&h_input, bytes, cudaHostAllocMapped);
cudaHostAlloc((void**)&h_output, bytes, cudaHostAllocMapped);

cudaHostGetDevicePointer((void**)&d_input, h_input, 0);
cudaHostGetDevicePointer((void**)&d_output, h_output, 0);

RemapKernel << <grid, block >> > (d_input, d_output, width, height, pattern);
cudaDeviceSynchronize();
```

그림 4. 메모리 방식별 핵심 코드 구성

Fig. 4. Core Code Structure by Memory Method

연속 접근은 연속된 메모리 주소를 읽는 경우를, 국소 비연속 접근은 일정한 오프셋을 가진 주변 데이터를 참조하는 경우를, 분산 접근은 떨어진 위치의 데이터를 읽는 불규칙한 접근 형태를 의미한다[7].

표 3. 실험 조건

Table 3. Experimental Conditions

구분	조건
워크로드	대규모 격자 재표본화
흐름 구성	CPU → GPU → CPU → GPU → CPU Read
접근 패턴	연속 접근(linear) / 국소 비연속 접근(local-offset) / 분산 접근(scatter)
메모리 방식	명시적 복사 방식(Device Memory) / 통합 메모리 방식(Unified Memory) / 제로 카피 방식(Zero-copy)
격자 크기	16000×16000 및 대용량 격자
실행 방식	cold-start, 10회 반복 평균, 표준편차 적용

www.kci.go.kr

각 조건은 cold-start 방식으로 10회 반복 수행하였다. 결과는 평균과 표준편차로 제시하고, 반복 측정값을 이용해 계산하였다.

3.4 측정 항목 및 분석 방법

표 4는 측정 항목을 나타낸다. 전체 실행시간과 데이터 이동, 커널 실행, CPU 결과 읽기 구간을 함께 측정하였다.

표 4. 성능 측정 항목
Table 4. Performance Measurement Items

항목	의미
MoveAvg	복사 또는 prefetch 구간
GPUAvg	GPU 처리 구간 시간
CPUAvg	CPU 처리 및 결과 읽기
ExecAvg	전체 실행 시간
RemapKernel	실제 CUDA 커널 시간
PCIe RX/TX	Host-GPU 전송 변화
L1/L2 Hit Rate	캐시 효율
Memory Throughput	GPU 커널 수행 중 메모리 계층 전체의 처리율

Nsight Systems를 이용하여 데이터 이동과 커널 실행 흐름을 확인하였다[17]. 이를 통해 메모리 복사 구간과 커널 수행 구간의 비중을 비교하고, 각 메모리 방식에서 데이터 이동이 전체 실행시간에 미치는 영향을 분석하였다. PC 환경에서는 nvidia-smi dmon을 사용하여 PCIe 수신 및 송신 대역폭을 측정하였다. 또한 PC 환경의 명시적 복사 방식에서 연속 접근과 분산 접근을 대상으로 Nsight Compute를 수행하여 L1/TEX Hit Rate, L2 Hit Rate 및 Memory Throughput을 분석하였다[18]. 이를 통해 접근 패턴에 따른 캐시 활용도와 메모리 처리율 차이를 확인하였다.

4. 결과 및 분석

본 장에서는 PC 환경과 Jetson 환경에서 동일한 대규모 격자 처리 CUDA 코드를 실행하고, 메모리 방식과 접근 패턴에 따른 성능 차이를 분석하였다. 실행 시간은 전체 실행 시간 기준으로 정리하였다. 표 5부터 표 9까지의 모든 측정값은 동일 조건에서 10회 반복 수행한 후 평균과 표준편차를 계산하여 제시하였다. 접근 패턴은 연속, 국소 비연속, 분산으로 구분하였다.

4.1 PC 환경의 메모리 방식별 실행 결과

표 5는 PC 이산형 GPU 환경에서 메모리 방식과 접근 패턴에 따른 전체 실행 시간을 비교한 결과이다.

표 5. PC 환경의 메모리 방식별 실행 결과
Table 5. Execution Results by Memory Method in the PC Environment

접근 패턴	명시적 복사 방식 (ms)	통합 메모리 방식 (ms)	제로 카피 방식 (ms)
연속	602.135 ± 34.632	10178.745 ± 132.334	637.524 ± 49.252
국소 비연속	584.794 ± 9.179	10126.0 ± 188.698	812.3 ± 50.589
분산	811.379 ± 18.678	10053.1 ± 87.478	6828.3 ± 29.379

PC 이산형 GPU 환경에서는 복사 비용을 줄이는 방식이 항상 유리하지 않으며, GPU 전용 메모리를 직접 사용하는 명시적 복사 방식(Device Memory)이 가장 안정적인 성능을 보였다. 제로 카피 방식(Zero-copy)은 분산 접근에서 host mapped memory의 비규칙 참조 부담이 커지며

www.kci.go.kr

성능 저하가 크게 나타났다. 통합 메모리 방식(Unified Memory)은 페이지 이동과 CPU-GPU 접근 전환 비용이 누적되어 전반적인 실행 시간 증가로 이어진 것으로 해석된다. 그 원인은 다음 절에서 구간별 특정 결과를 통해 분석하였다.

4.2 PC Unified Memory 성능 저하 분석

표 6은 연속 조건에서 통합 메모리 방식의 성능 저하가 어느 구간에서 발생하는지 확인하기 위해 세부 시간과 프로파일링 지표를 비교한 결과이다.

표 6. PC 환경의 Unified Memory 성능 저하 발생 위치 분석

Table 6. Analysis of Unified Memory Performance Degradation in the PC Environment

분석 항목	명시적 복사 방식	통합 메모리 방식
ExecAvg (ms)	602.135 ± 34.632	10178.745 ± 132.334
GPUAvg (ms)	8.921 ± 0.284	12.216 ± 0.255
CPUAvg (ms)	0.064 ± 0.002	1348.329 ± 4.497
실제 RemapKernel 시간(ms)	2.436 ± 0.011	2.435 ± 0.010
PCIe TX 평균 처리율(MB/s)	9.214 ± 0.183	286.735 ± 5.624

통합 메모리 방식에서 실제 RemapKernel 시간은 명시적 복사 방식과 거의 차이가 없었다. 따라서 성능 저하는 계산보다 페이지 이동과 CPU- GPU 간 데이터 소유권 전환 과정에서 발생한 것으로 볼 수 있다.

CPU 결과 접근 시간과 PCIe TX의 증가는 GPU 처리 후 CPU가 결과 데이터에 다시 접근하는 과정에서 데이터 이동과 동기화 부담이 증가했음을 시사한다. 다만 PCIe TX는 페이지 이동량을 직접 측정한 값은 아니므로, 통합 메모리 방식에서 발생한 암묵적 데이터 이동을 뒷받침하는 보조 지표로 해석하였다. 전체 실행 시간 차이에는 CPU 결과 접근 외에도 GPU 최초 접근 시의 페이지 이동, 초기화 및 접근 주체 전환 비용이 함께 포함된다.

4.3 Jetson 환경의 메모리 방식별 실행 결과

표 7은 Jetson AGX Orin 환경에서 동일한 조건으로 측정한 전체 실행 시간이다.

표 7. Jetson 환경의 메모리 방식별 실행 결과

Table 7. Execution Results by Memory Method in the Jetson Environment

접근 패턴	명시적 복사 방식	통합 메모리 방식	제로 카피 방식
연속	1246.310 ± 11.464	759.053 ± 72.689	746.625 ± 66.583
국소 비연속	1255.350 ± 8.606	760.826 ± 66.047	755.249 ± 63.392
분산	15100.300 ± 31.854	14591.300 ± 146.406	14575.600 ± 57.040

Jetson에서는 연속과 국소 비연속에서 통합 메모리 방식과 제로 카피 방식이 명시적 복사 방식보다 낮은 실행 시간을 보였다. CPU와 GPU가 물리 메모리를 공유하는 UMA 구조에서는 별도 복사와 중복 버퍼를 유지하는 부담이 상대적으로 크게 작용한 결과로 해석된다.

반면 분산 접근에서는 세 방식 모두 실행 시간이 크게 증가하였다. 이는 Jetson에서도 메모리 방식보다 접근 패턴 자체가 성능을 제한할 수 있음을 보여준다.

www.kci.go.kr

4.4 Jetson 환경의 대용량 linear 처리 결과

표 8은 Jetson 환경에서 연속 접근 패턴 조건으로 격자 크기를 증가시키며 명시적 복사 방식과 제로 카피 방식의 실행 가능성을 비교한 결과이다.

표 8. Jetson 환경의 대용량 linear 처리 결과
Table 8. Large-scale Linear Processing Results in the Jetson Environment

격자 크기	명시적 복사 방식	제로 카피 방식
20000×20000	1982.912 ± 24.643	1634.359 ± 19.824
30000×30000	4212.815 ± 41.342	3721.309 ± 35.724
40000×40000	OOM	6623.323 ± 58.442

Jetson과 같은 UMA 구조에서는 CPU와 GPU가 같은 물리 메모리를 사용한다. 그러나 명시적 복사 방식은 Host buffer와 Device buffer를 따로 만들어 데이터를 한 번 더 복사해야 하므로 메모리 사용량이 증가한다. 대용량 조건에서는 이러한 중복 버퍼 때문에 메모리 부족이 발생할 수 있다. 반면 제로 카피 방식은 별도의 Device buffer를 만들지 않아 추가 복사가 필요 없으며, 메모리 사용 부담이 작아 동일한 대용량 조건에서도 실행 가능성을 확보할 수 있다.

4.5 접근 패턴에 따른 커널 병목 분석

표 9는 명시적 복사 방식 기준으로 연속 접근 패턴과 분산 접근 패턴에서 측정한 캐시 효율 및 메모리 처리율을 비교한 결과이다.

PC와 Jetson 모두 분산 접근에서 L1/TEX Hit Rate와 L2 Hit Rate가 감소하였다. 인접 스레드가 서로 다른 위치를 참조하면서 캐시 재사용이 줄어든 결과이다. 특히 Jetson에서는 분산 접근 시 L2 Hit Rate와 Memory Throughput이 크게 낮아졌다. UMA 구조라도 비규칙 접근이 포함되면 커널 내부의 메모리 처리 효율이 크게 떨어질 수 있음을 확인하였다.

표 9. 접근 패턴별 커널 메모리 효율 분석 결과
Table 9. Kernel Memory Efficiency Analysis by Access Pattern

환경	접근패턴	L1/TEX Hit Rate (%)	L2 Hit Rate (%)	Memory Throughput (%)
PC	연속	45.139 ± 0.003	66.442 ± 0.249	82.455 ± 0.128
PC	분산	38.400 ± 0.005	11.360 ± 0.107	30.802 ± 0.013
Jetson	연속	45.513 ± 0.008	66.562 ± 0.409	40.802 ± 0.380
Jetson	분산	38.980 ± 0.012	11.440 ± 3.376	0.863 ± 0.016

따라서 대규모 격자 처리에서는 메모리 방식 선택뿐만 아니라, 입력 참조 위치를 최대한 연속화하거나 분산 접근(scatter)을 줄이는 접근 패턴 최적화가 함께 필요하다.

5. 결론 및 향후 연구

본 연구에서는 PC 이산형 GPU 환경에서 작성한 대규모 격자 재표본화 CUDA 코드를 Jetson UMA GPU 환경으로 이식할 때, 메모리 방식과 접근 패턴에 따른 실행 특성을 비교하였다. 표 10은 본 실험 범위에서 관찰된 결과를 바탕으로 메모리 방식 검토 방향을 정리한 것이다.

PC 이산형 GPU 환경에서는 명시적 복사 방식이 가장 안정적인 실행 성능을 보였다. 통합 메모리 방식은 CPU-GPU 접근 전환과 페이지 관

www.kci.go.kr

리 비용의 영향을 크게 받았으며, 제로 카피 방식은 분산 접근에서 호스트 메모리 직접 접근 비용으로 성능 저하가 나타났다.

표 10. CUDA 메모리 방식 실험 가이드라인
Table 10. CUDA Memory Method Experimental Guideline

플랫폼	처리 조건	우선 검토 방식
PC 이산형 GPU	일반 격자 처리	명시적 복사 방식(Device Memory)
PC 이산형 GPU	연속 접근 또는 국소 비연속 접근 중심	명시적 복사 방식을 기준으로 비교
PC 이산형 GPU	분산 접근 포함	명시적 복사 방식 및 접근 패턴 최적화
Jetson UMA GPU	일반 조건	통합 메모리 방식 및 제로 카피 방식 비교
Jetson UMA GPU	코드 이식성 우선	통합 메모리 방식(Unified Memory) 검토
Jetson UMA GPU	대용량 데이터 또는 메모리 여유 부족	제로 카피 방식(Zero-copy) 검토
Jetson UMA GPU	분산 접근 포함	메모리 방식 비교와 접근 패턴 최적화 병행

Jetson UMA 환경에서는 CPU와 GPU가 물리 메모리를 공유하므로, 대용량 조건에서 호스트 버퍼와 장치 버퍼를 함께 유지하는 명시적 복사 방식의 메모리 부담이 커질 수 있었다. 반면 제로 카피 방식은 별도 장치 버퍼 구성을 줄일 수 있어 대용량 연속 접근 조건에서 메모리 운용 측면의 장점을 보였다. 통합 메모리 방식은 기존 코드의 변경을 줄일 수 있으나, 접근 주체 전환이 반복되는 조건에서는 성능 변화를 함께 확인할 필요가 있다.

분산 접근에서는 두 플랫폼 모두 캐시 효율과 메모리 처리율이 감소하였으므로, 메모리 방식 변경만으로 성능 문제를 해결하기 어렵다. 따라서 실제 이식 과정에서는 플랫폼 구조, 데이터 크기, CPU-GPU 접근 순서와 함께 접근 패턴 최적화를 병행해야 한다. 본 지침은 본 연구의 격자 재표본화 실험 범위에 한정되며, 향후 다양한 대규모 데이터 처리 응용에서 좌표변환, 보간, 재표본화 등이 결합된 조건에 대해 추가적인 검증이 필요하다.

참 고 문 헌

[1] Liu, S., Liu, L., Tang, J., Yu, B., Wang, Y., Shi, W., "Edge Computing for Autonomous Driving: Opportunities and Challenges", Proceedings of the IEEE, Vol.107, No.8, pp.1697-1716, 2019, DOI : 10.1109/JPROC.2019.2915983

[2] Satyanarayanan, M., "The Emergence of Edge Computing", Computer, Vol.50, No.1, pp.30-39, 2017, DOI : 10.1109/MC.2017.9

[3] Shi, W., Cao, J., Zhang, Q., Li, Y., Xu, L., "Edge Computing: Vision and Challenges", IEEE Internet of Things Journal, Vol.3, No.5, pp.637-646, 2016, DOI : 10.1109/JIOT. 2016.2579198

[4] Zhang, F., Hu, C., Li, W., Hu, W., Li, H. C., "Accelerating Time-Domain SAR Raw Data Simulation for Large Areas Using Multi-GPUs", IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, Vol.7, No.9, pp.3956-3966, 2014, DOI : 10.1109/JSTARS.2014.2330333

[5] Sanders, J., Kandrot, E., "CUDA by Example: An Introduction to General-Purpose GPU Programming", Addison-Wesley, 2010,

www.kci.go.kr

ISBN : 978-0131387683
[6] NVIDIA, “CUDA C++ Programming Guide (Legacy)”, NVIDIA Documentation, 2025, https://docs.nvidia.com/cuda/cuda-c-programming-guide/
[7] Wong, H., Papadopoulou, M. M., Sadooghi-Alvandi, M., Moshovos, A., “Demystifying GPU Microarchitecture through Microbenchmarking”, IEEE ISPASS, IEEE, pp. 235-246, 2010, DOI: 10.1109/ISPASS.2010.5452013.
[8] Shin, D. J., Kim, J. J., “A Deep Learning Framework Performance Evaluation to Use YOLO in NVIDIA Jetson Platform”, Applied Sciences, Vol.12, No.8, Article 3734, 2022, DOI : 10.3390/app12083734
[9] Mittal, S., “A Survey on Optimized Implementation of Deep Learning Models on the NVIDIA Jetson Platform”, Journal of Systems Architecture, Vol.97, pp.428-442, 2019, DOI : 10.1016/j.sysarc.2019.01.011
[10] Lema, D. G., Usamentiaga, R., García, D. F., “Quantitative Comparison and Performance Evaluation of Deep Learning-Based Object Detection Models on Edge Computing Devices”, Integration, Vol.95, Article 102127, 2024, DOI : 10.1016/j.vlsi.2023.102127
[11] NVIDIA, “NVIDIA Jetson AGX Orin Series Technical Brief”, NVIDIA Technical Brief, 2022, https://www.nvidia.com/content/dam/en-zz/Solutions/gtcf21/jetson-orin/nvidia-jetson-agx-orin-technical-brief.pdf
[12] Yu, Q., Childers, B. R., Huang, L., Qian, C., Wang, Z., “A Quantitative Evaluation of Unified Memory in GPUs”, The Journal of Supercomputing, Vol.76, pp.2958-2985, 2020, DOI : 10.1007/s11227-019-03079-y
[13] Arauzo, X., et al., “Unraveling the Mystery of NVIDIA's Unified Memory for Safety-Critical GPU Systems”, In 2023 26th Euromicro Conference on Digital System Design (DSD), IEEE, pp.366-372, 2023, DOI : 10.1109/DSD60849.2023.00058
[14] Li, W., Jin, G., Cui, X., See, S., “An Evaluation of Unified Memory Technology on NVIDIA GPUs”, In 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), IEEE, pp.1092-1098, 2015, DOI : 10.1109/CCGrid. 2015.105
[15] NVIDIA, “NVIDIA Ada GPU Architecture”, NVIDIA Whitepaper, 2022, https:// images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf
[16] Ren, J., Yu, G., He, Y., Li, G. Y., “A Survey on End-Edge-Cloud Orchestrated Network Computing Paradigms: Transparent Computing, Mobile Edge Computing, Fog Computing, and Cloudlet”, ACM Computing Surveys, Vol.52, No.6, Article 125, 2019, DOI : 10.1145/3362031
[17] NVIDIA, “NVIDIA Nsight Systems User Guide”, NVIDIA Documentation, 2025, https://docs.nvidia.com/nsight-systems/
[18] NVIDIA, “NVIDIA Nsight Compute User Guide”, NVIDIA Documentation, 2025, https://docs.nvidia.com/nsight-compute/
[19] Luo, W., Fan, R., Li, Z., Du, D., Wang, Q., Chu, X., “Benchmarking and Dissecting the NVIDIA Hopper GPU Architecture”, In 2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, pp.656-667, 2024, DOI : 10.1109/IPDPS57955.2024.00064
[20] Cooper, B., Scogland, T. R. W., Ge, R., “Shared Virtual Memory: Its Design and Performance Implications for Diverse Applications”, In Proceedings of the 38th ACM International Conference on Supercomputing (ICS), ACM, pp.26-37, 2024, DOI : 10.1145/3650200.3656608

www.kci.go.kr

저 자 소 개

김민수(Min-Soo Kim)

2025.2 우송대학교 컴퓨터정보보안학과 학사
2025.3-현재 국립공주대학교 컴퓨터공학과 석사 과정
<주관심분야> 소프트웨어 공학, 인공지능, 항공위성, 빅데이터

김재웅(Jae-Woong Kim)

1983.2 중앙대학교 전자계산학과 학사
1988.2 중앙대학교 컴퓨터공학과 석사
2002.2 대전대학교 컴퓨터공학과 박사
1992.8-현재 국립공주대학교 천안공과대학 컴퓨터공학부 교수
<주관심분야> 소프트웨어공학, 인공지능 시스템, 멀티미디어공학, 빅데이터

장미영(Mi-Young Jang)

2003.2 배재대학교 유전공학과 학사
2024.8 청주교육대학교 초등정보&로봇과 석사
2025.3-현재 국립공주대학교 컴퓨터공학과 박사 과정
<주관심분야> 소프트웨어 공학, 인공지능, 항공위성, 빅데이터

www.kci.go.kr