HPC 환경에서 Athena++ 시뮬레이션 코드 GPU 병렬화 및 성능 최적화 연구

(A Study on GPU Parallelization and Performance Optimization of the Athena++ Simulation Code in High-Performance Computing Environments)

정현미*, 이현조**, 정기문*, 채철주**

(Hyun Mi Jung, Hyunjo Lee, Kimoon Jeong, Cheol-Joo Chae)

요 약

본 논문에서는 자기유체역학 시뮬레이션 코드인 Athena++의 핫스팟을 식별하고, 고성능컴퓨팅 환경에서의 연산 가속을 위한 병렬처리 최적화 기법을 제안한다. 코드 프로파일링 도구(gprof, valgrind, vtune)를 활용하여 주요 연산 핫스팟은 Hydro::RiemannSolver 부분임을 확인하였으며, 이를 대상으로 CUDA 기반 GPU 병렬화를 수행하였다. 제안 방법의 성능 평가는 Google Colab의 A100 GPU 환경에서 수행되었으며, 단일 연산 사이클 기준 평균 실행 시간이 기존 CPU 대비 약 25배 이상 향상되었고, 반복 연산 구조의 병렬처리를 통해 연산 효율이 크게 개선되었음을 확인하였다. 이러한 결과는 이기종 CPU-GPU 환경에서의 병렬화가 고성능 시뮬레이션 코드의 실질적인 성능 향상에 크게 기여할 수 있음을 시사한다. 또한, 차세대 컴퓨팅 디바이스(CXL, DPU 등)에 적합한 아키텍처 설계와 데이터 흐름 최적화 전략이 향후 HPC 응용 프로그램 성능 개선의 핵심 요소가 될 것임을 확인할 수 있다.

■ 중심어 : 고성능컴퓨팅 ; Athena++; 자기유체역학 ; 리만 솔버 ; 코드 프로파일링

Abstract

This paper identifies performance hotspots in the magnetohydrodynamics (MHD) simulation code Athena++ and proposes parallel optimization techniques for computational acceleration in high-performance computing (HPC) environments. Through code profiling tools such as gprof, valgrind, and vtune, the primary computational hotspot was found to be the Hydro::RiemannSolver module. Performance evaluation was conducted in the Google Colab environment using the A100 GPU. The results demonstrated over a 25-fold improvement in average execution time per computation cycle compared to the CPU implementation, confirming substantial enhancement in computational efficiency through parallel processing of repetitive structures. These findings suggest that parallelization in heterogeneous CPU-GPU environments can significantly improve the performance of high-fidelity simulation codes. Moreover, architecture design and data flow optimization tailored for next-generation computing devices such as CXL and DPUs are expected to play a critical role in future HPC application performance improvement.

■ keywords : High-Performance Computing; Athena++ ; Magnetohydrodynamics ; Riemann Solver ; Code Profiling

Ⅰ. 서 론

HPC(High Performance Computing)는 복잡한 과학 및 공학 문제의 수치 해석을 위한 핵심 인프라로 최근에는 인공지능, 유체역학, 화학, 생명공학 등 다양한 응용 분야에서 활용되고 있다. HPC 응용 어플리케이션은 HPC 인프라를 이용하여 복잡하고

* 정회원, 한국과학기술정보연구원 슈퍼컴퓨팅기술개발센터
** 정회원, 한국농수산대학교 교양학부
이 논문은 2025년도 한국과학기술정보연구원(KISTI)의 기본사업으로 수행된 연구입니다.(과제번호:K25L1M2C2)

접수일자 : 2025년 06월 09일
게재확정일 : 2025년 06월 16일
교신저자 : 채철주 e-mail : chae.cheoljoo@gmail.com

방대한 계산을 요구하는 문제를 해결하는 소프트웨어로 시간적, 공간적 고차원으로 구성된 대규모 데이터를 기반으로 하는 계산 집약적 응용에 많이 사용되고 있다. 이러한 HPC 응용 어플리케이션에서는 대규모 데이터를 이용하기 때문에 연산 오버헤드와 데이터 이동 병목 현상의 해결이 가장 큰 난제이다. 이에 따라 MPI, OpenMP, CUDA 등 다양한 병렬 프로그래밍 모델이 실제 응용 코드에 통합되고 있으며, CPU와 GPU를 연계한 하이브리드 아키텍처도 활발히 연구되고 있다[1-5].

그러나 단지 GPU를 적용하는 것을 넘어, CPU-GPU 이기종 클러스터 환경에서 작업의 특성, 데이터 전송, 스케줄링 정책 등 다양한 요인이 전체 시스템 성능에 미치는 영향을 체계적으로 분석하고 최적화하는 연구가 필요하다. 또한 GPU를 이용한 연산에서 비용 등의 한계점 극복을 위하여 차세대 디바이스 (CXL(Compute Express Link), DPU(Data Process Unit) 등)를 활용한 이기종 HPC 시스템 기반의 연산 방안연구가 필요하다.

본 논문에서는 HPC 기반 병렬 처리 가속 기술을 이용한 성능 개선 연구를 분석한다. 또한 HPC 응용 적용 실험을 통해 성능 향상을 검증하고, 본질적으로 병렬화가 필요한 HPC 응용 연산에 대한 가속 전략을 제안한다. 이를 위하여, 천체 물리학 자기유체역학 시뮬레이션인 Athena++의 병렬 처리 구조를 최적화하고, 핵심 연산 모듈의 성능 향상 방안을 제시한다. 실험을 위하여 Athena++에 내장된 주요 Solver의 구조 및 기능을 분석하고, 내부 데이터 계층과 함수 호출 흐름에 대한 정밀 분석을 통해 병렬화에 적합한 연산 단위를 식별하였으며, 프로파일링 도구를 이용하여 병목 지점을 분석하였다. 분석 결과를 통하여 성능 향상을 위한 CPU-GPU 기반 이기종 HPC 아키텍처에서의 분산 병렬처리 방안을 제안한다. 향후 분석 결과를 이용하여 다양한 차세대 디바이스 기반의 이기종 HPC 아키텍처에의 적용을 통해 성능 향상의 문제점을 해결하며. 이를 통해 보다 실질적이고 유용한 기반을 제공하고자 한다.

본 논문의 구성은 다음과 같다. 2장에서는 HPC 기반 병렬 처리 가속 기술을 이용한 성능 개선 연구를 분석한다. 3장에서는 Athena++ 유체역학 수치해석을 위한 알고리즘에 대해 분석하고 코드 프로파일 도구를 이용하여 핫스팟을 분석하고 성능 개선 방안을 제안한다. 4장에서는 GPU 기반 분산병렬처리 코드 변환 및 성능을 테스트하고 5장에서 결론을 맺는다.

Ⅱ. 관련연구

병렬처리 기술을 활용한 HPC 응용프로그램의 성능 개선을 위해 병렬처리 기술을 활용한 연구는 다양한 분야에서 활발히 진행되고 있다. 이번 장에서는 대표적인 병렬 프로그래밍 모델인 MPI, OpenMP와 GPU 가속을 위한 CUDA를 적용하여 계산 시간을 단축하고 효율성을 극대화한 주요 연구를 분석한다.

1. MPI 및 OpenMP 기반 병렬처리 연구

MPI(Message Passing Interface)는 분산 메모리 시스템(클러스터)의 노드 간 통신을, OpenMP는 공유 메모리 시스템(단일 노드 내 멀티코어)의 스레드 간 협업을 지원한다. 이 두 기술을 단독 또는 하이브리드 형태로 사용하여 성능을 개선한 연구들이 다수 존재한다. “다물체 페리다이나믹 해석을 위한 MPI-OpenMP 혼합 병렬화” 에서는 고체 역학 및 파괴 시뮬레이션 (Peridynamics)의 복잡하고 계산량이 많은 다물체 동적 파괴 현상 해석을 위해 Fortran 기반 코드를 MPI-OpenMP 하이브리드 모델로 병렬화하였다.

KISTI 슈퍼컴퓨터 5호기(누리온)에서 성능을 검증하였다. 그 결과 노드 내에서는 OpenMP로 스레드를 할당하고, 노드 간에는 MPI로 통신하는 하이브리드 방식이 순수 MPI 방식보다 우수한 성능 확장성(scalability)을 보임을 입증하였다. 이는 통신 오버헤드를 줄이고 계산 자원을 효율적으로 활용한 결과이다[6].

“Optimizing the Weather Research and Forecasting Model with OpenMP Offload” 에서는 기상 예측 (Weather Research and Forecasting,

WRF 모델)을 기존MPI와 OpenMP(CPU)로 병렬화되었던 WRF 모델의 계산 병목 구간(미세물리 스킴)을 OpenMP Offload 지시어를 사용해 GPU로 가속하였다. 이를 미국 NERSC의 Perlmutter 슈퍼컴퓨터에서 테스트한 결과, 뇌우 시뮬레이션 사례에서 전체 모델 수행 속도가 2.08배 향상되었다. 이는 특정 계산 집약적 커널을 GPU로 이전하여 얻은 성과이다[7].

“OpenMP와 MPI 코드의 상대적, 혼합적 성능 고찰 ”에서는 SPEC HPC2002 벤치마크 하여, 클러스터 및 SMP(Symmetric Multi-Processing) 환경에서 동일한 응용프로그램을 순수 MPI, 순수 OpenMP, 그리고 MPI-OpenMP 하이브리드 방식으로 각각 구현하여 성능을 비교 분석하였다. 그 결과 응용프로그램의 특성과 하드웨어 아키텍처에 따라 최적의 병렬화 모델이 다르다는 것을 실증하였다. 일반적으로 클러스터 환경에서는 하이브리드 모델이 통신과 계산을 효율적으로 분배하여 높은 성능을 제공함을 확인하였다[8].

2. GPU 가속(CUDA)을 통한 성능 개선 연구

CUDA(Compute Unified Device Architecture)는 NVIDIA GPU의 대규모 병렬 프로세서를 활용하여 계산 집약적인 부분을 고속으로 처리하는 기술이다. 특히 분자 동역학, 유체 역학, 딥러닝 등의 분야에서 혁신적인 성능 개선을 이끌고 있다. “Task-parallelism in SWIFT for heterogeneous compute architectures”에서는 CPU-GPU 이기종 시스템에서 우주론적 유체 동역학 시뮬레이션(Smoothed Particle Hydrodynamics, SPH) 코드(SWIFT)의 성능을 최적화하였다. 기존의 태스크 기반 병렬 구조를 활용하여, 메모리 접근이 많은 계산은 CPU가, 계산량이 많은 연산(입자 간 상호작용)은 GPU가 동시에 처리하도록 작업을 분배하였다. 이를 통하여 CPU-GPU 간의 데이터 통신 지연을 최소화하는 알고리즘을 통해 이기종 엑사스케일 시스템의 하드웨어 자원을 최대한 활용할 수 있는 가능성을 제시하였다[9].

“Comparative Study of the Parallelization of the Smith-Waterman Algorithm on OpenMP and Cuda C” 에서는 생명정보학(Bioinformatics) 서열정렬(Smith-Waterman 알고리즘)중 단백질이나 DNA 서열의 유사성을 찾는 Smith-Waterman 알고리즘을 OpenMP와 CUDA C를 사용하여 각각 병렬화하고 성능을 비교하였다. 데이터 종속성을 극복하기 위해 행렬의 반대각선(anti-diagonal) 구조를 활용하였다. 이 결과 순차적 알고리즘 대비 OpenMP와 CUDA 모두에서 상당한 속도 향상을 보였으며, 특히 대규모 병렬 처리가 가능한 CUDA 구현이 더 높은 가속 성능을 나타냈다[10].

Ⅲ. Athena++ 유체역학 수치해석 알고리즘 및 핫스팟 분석

Athena++는 C++로 작성된 천체 물리학 자기유체역학 시뮬레이션으로 AMR(Advanced Mesh Refinement)을 활용하여 유연한 좌표 및 그리드 설정을 지원한다[11-12]. Athena++ 내 구현되어 있는 대표적인 Solver로는 HLL(Harten-Lax-van Leer)와 ROE(Roe’s solver)가 있다. HLL은 대표적인 근사 리만 해법으로 불연속(x=0)의 왼쪽과 오른쪽 영역에서 최대 및 최소 파장속도(파속)을 측정하여 이를 바탕으로 좌영역의 파장과 우영역의 파장 사이의 Star region에서 발생하는 파장의 속도, 압력, 밀도 등을 근사계산한다[13]. 이를 바탕으로 다양한 확장 근사 리만 해법이 제안되었다. ROE는 Godunov 방식을 기반으로 하는 근사 리만 해법으로 이산화된 시공간 영역에서 두 셀 사이의 Godunov 수치 유량(Flux)에 대한 추정치를 계산하는 방법이다. 표 1은 Athena++ 내 구현된 Solver 형태 및 특징에 대해 보여주고 있으며, 그림 1은 3D MHD 시뮬레이션 실행 결과를 보여주고 있다.

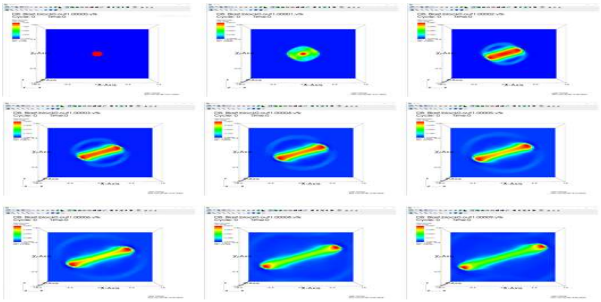


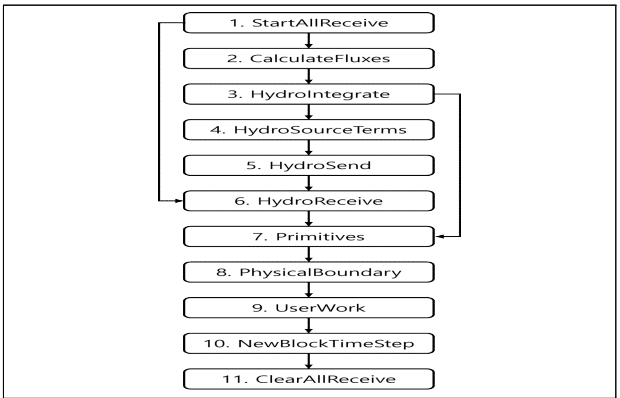
그림 1 Athena++ 3D MHD 시뮬레이션 실행 결과

표 1. Athena++ Solver 개요

Solver Type	Solver Name	개요
Hydrodynamic	Harten-Lax-van Leer-Contact	HLL에 접촉파를 고려하기 위해 중앙 파속 계산을 추가
	Harten-Lax-van Leer and Einfeldt	음의 밀도나 압력을 반환하지 않는 HLL 기반 해법
	Local Lax Friedrichs	반대 방향에서 서로 다른 속도의 두 파장이 발생할 때를 가정하는 가장 단순한 해법
	Low-dissipation HLLC	저속 충격파를 위한 HLL 기반 해법
	Roe's solver	ROE 근사 리만 해법
MHD	Harten-Lax-van Leer-discontinuities	알펜파를 고려한 HLL 기반 해법
	HLLE-MHD	HLLE 상에서 알펜파 고려하도록 수정한 근사 리만 해법
	Low-dissipation HLLD	저속 자기유체파장을 위한 HLLD 기반 해법
	LLF-MHD	알펜파를 고려한 LLF
	ROE-MHD	알펜파를 고려한 ROE 근사 리만 해법

Athena++에서 Hydrodynamic에서는 HLLC solver를 추천하고 MHD에서는 HLLD solver를 추천하고 있다. 표 2와 표 4는 HLLC와 HLLD를 이용한 유동체 유량 측정 흐름 분석 결과를 보여주고 있다.

표 2. HLLC를 이용한 유동체 유량 측정 흐름도



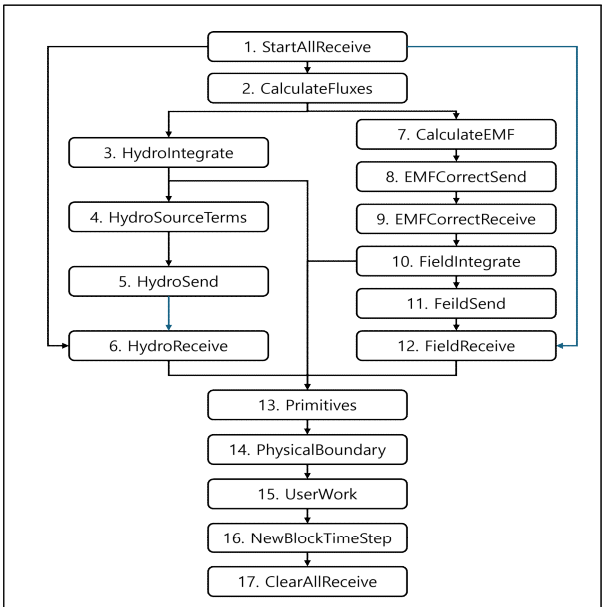
단계 이름	설명
1. StartAllReceive	전체 데이터 입력
2. CalculateFluxes	유량 계산
3. HydroIntegrate	계산된 유량 적분
4. HydroSourceTerms	유체 흐름에 의해 영향받는 기하학적 요인을 현 좌표에 맞게 재설정
5. HydroSend	유체 정보 송신
6. HydroReceive	유체 정보 수신
7. Primitives	유체 원시 정보 관리
8. PhysicalBoundary	도메인, 그리드, 셀의 물리 경계 재설정
9. UserWork	사용자 정의 함수 처리
10. NewBlockTimeStep	다음 작업을 위한 블록 생성
11. ClearAllReceives	작업 마무리

HLLC solver 호출 순서는 Main→TaskList→AddTask→TaskIntegrate→CalculateHydroFlux→CalculateFluxes이며, 호출 대상 및 역할은 표 3에서 보여주고 있다.

표 3. HLLC Solver 호출 흐름

순서	호출자	호출 대상	역할
1	Main	TaskList	새로운 문제 Task 생성
2	TaskList	AddTask	생성된 Task를 리스트에 추가
3	AddTask	TaskIntegrate	문제에 따른 Integrate 방법 설정
4	TaskIntegrate	CalculateHydroFlux	지정한 유량 계산 함수 설정
5	CalculateHydroFlux	CalculateFluxes	유량 계산 함수
6	CalculateFluxes	HLLC	유량 계산을 위한 근사 리만 해법

표 4. HLLD를 이용한 유동체 유량 측정 흐름도



단계 이름	설명
1. StartAllReceive	전체 데이터 입력
2. CalculateFluxes	유량 계산
3. HydroIntegrate	계산된 유량 적분
4. HydroSourceTerms	유체 흐름에 의해 영향 받는 기하학적 요인을 현 좌표에 맞게 재설정
5. HydroSend	유체 정보 송신
6. HydroReceive	유체 정보 수신
7. CalculateEMF	전자기장 영역 계산
8. EMFCorrectSend	(그리드별) 수정 전자기장 영역 송신
9. EMFCorrectReceive	(그리드별) 수정 전자기장 영역 수신
10. FieldIntegrate	전자기 유체 영역 적분
11. FieldSend	전자기 유체 영역 송신
12. FieldReceive	전자기 유체 영역 수신
13. Primitives	유체 원시 정보 관리
14. PhysicalBoundary	도메인, 그리드, 셀의 물리 경계 재설정
15. UserWork	사용자 정의 함수 처리
16. NewBlockTimeStep	다음 작업을 위한 블록 생성
17. ClearAllReceives	작업 마무리

HLLD solver 호출 순서는 Main→TaskList→AddTask→TaskIntegrate→CalculateHydroFlux→CalculateFluxes이며, 호출 대상 및 역할은 표 5에서 보여주고 있다.

표 5. HLLD Solver 호출 흐름

순서	호출자	호출 대상	역할
1	Main	TaskList	새로운 문제 Task 생성
2	TaskList	AddTask	생성된 Task를 리스트에 추가
3	AddTask	TaskIntegrate	문제에 따른 Integrate 방법 설정
4	TaskIntegrate	CalculateHydroFlux	지정한 유량 계산 함수 설정
5	CalculateHydroFlux	CalculateFluxes	유량 계산 함수
6	CalculateFluxes	HLLD	유량 계산을 위한 근사 리만 해법

Athena++ 데이터 구조는 Mesh, Domain, Grid로 구성되어 있으며, Mesh 객체 전체 연결에 대한 구조는 그림 2와 같다.

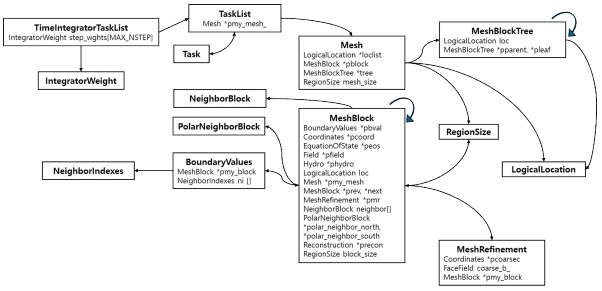


그림 2. Athena++ Mesh 객체 전체 연결

Athena++ 응용 어플리케이션의 효율성 향상을 위해 본 논문에서는 colab 환경에서 gprof, valgrind, vtune을 이용하여 코드 프로파일을 수행하였다. 코드 프로파일링은 비효율적인 코드 경로, CPU나 메모리를 많이 소비하는 함수를 식별하여, 최적화 작업의 우선순위를 결정하고 중요한 문제를 해결하는 단서를 획득할 수 있기 때문에 소프트웨어의 전반적인 성능을 개선하는데 효과적이다. 메모리와 CPU 등 리소스 사용량을 모니터링하여, 메모리 누수, 비효율적인 메모리 할당, 높은 CPU 사용량을 감지할 수 있다. 이를 통해 리소스 관리 효율을 향상시킬 수 있으므로 리소스 고갈로 인한 애플리케이션 충돌이나 실행 속도 저하 등의 문제를 완화할 수 있다. 코드 프로파링을 수행하였을 때 가장 많은 CPU 사용 시간을 소비하고 있는 부분은 Hydro::RiemannSolver가 최고의 핫스팟으로 확인되었으며, 두 번째 핫스팟은 EquationOfState::FastMagnetosonicSpeed으로 확인되었다. Hydro::RiemannSolver인 HLLD의 경우 "데이터 로드→for문을 통한 반복처리→결과 데이터 저장→반환"으로 이루어져 있으며 이는 병렬처리 구문으로 처리하기에 적합하지만 EquationOfState::FastMagnetosonicSpeed의 경우, 주어진 두 값에 대한 간단한 연산들을 수행하고 최종적으로 연산을 통해 획득한 값들에 대한 제곱근을 구하는 함수로 병렬처리를 수행하고자 하는 경우 해당 함수를 호출하는 모든 부분에서 코드 변환을 수행해야 하기 때문에 성능향상 개선 대상으로 부적합하다. 표 6-8은 코드 프로파링링 분석 결과를 보여주고 있다.

표 6. gprof 상위 10개 항목 출력 결과

No	cumulative	self	self	total		name
	sec	sec	calls	ms/call	ms/call	
1	75.68	75.68	8408400	0.01	0.01	Hydro::RiemannSolver
2	91.55	15.87	1360622832	0	0	EquationOfState::FastMagnetosonicSpeed
3	99.55	8	1415700	0.01	0.01	Reconstruction::PiecewiseLinearX3
4	106.45	6.9	1415700	0	0	Reconstruction::PiecewiseLinearX1
5	113.29	6.84	1415700	0	0	Reconstruction::PiecewiseLinearX2
6	119.74	6.45	650	9.92	9.92	Field::ComputeCornerE
7	125.77	6.03	650	9.28	10	Hydro::AddFluxDivergence
8	130.08	4.31	326	13.22	22.75	Hydro::NewBlockTimeStep
9	132.81	2.73	651	4.19	6.5	EquationOfState::ConservedToPrimitive
10	135.16	2.35	1415700	0	0	Reconstruction::DonorCellX3

표 7. vtune 상위 10개 항목 출력 결과

No	Function	CPU time			
		Total	Effective Time	Spin Time	Overhead
1	Hydro::RiemannSolver	69.086s	69.086s	0s	0s
2	EquationOfState::FastMagnetosonicSpeed	15.990s	15.990s	0s	0s
3	Reconstruction::PiecewiseLinearX3	6.768s	6.768s	0s	0s
4	Reconstruction::PiecewiseLinearX1	5.532s	5.532s	0s	0s
5	Reconstruction::PiecewiseLinearX2	5.041s	5.041s	0s	0s
6	Field::ComputeCornerE	4.940s	4.940s	0s	0s
7	Hydro::AddFluxDivergence	3.919s	3.919s	0s	0s
8	Hydro::NewBlockTimeStep	3.088s	3.088s	0s	0s
9	EquationOfState::ConservedToPrimitive	2.062s	2.062s	0s	0s
10	Reconstruction::DonorCellX3	1.910s	1.910s	0s	0s

표 8. valgrind 상위 10개 항목 출력 결과

No	# of calls	Ratio	Function
1	442,801,392,005	61.22%	Hydro::RiemannSolver
2	43,539,930,624	6.02%	EquationOfState::FastMagnetosonicSpeed
3	32,621,345,202	4.51%	Reconstruction::PiecewiseLinearX1
4	29,381,287,071	4.06%	Reconstruction::PiecewiseLinearX2
5	29,367,075,774	4.06%	Reconstruction::PiecewiseLinearX3
6	22,913,850,700	3.17%	Field::ComputeCornerE
7	22,148,977,500	3.06%	Hydro::AddFluxDivergence
8	20,941,726,876	2.90%	Hydro::NewBlockTimeStep
9	10,496,484,432	1.45%	EquationOfState::ConservedToPrimitive
10	10,128,673,100	1.40%	Field::CT

본 논문에서는 CUDA GPU 프로그래밍을 활용하여 HLLD에 대한 병렬처리를 수행하였으며, 데이터 로드 파트는 CPU 데이터를 GPU에 전송하기 위한 메모리 구조 생성 및 복사하였으며, for 반복문 파트는 GPU에서 병렬처리하였다. 반복문 내부에서 외부 함수 접근 시 외부함수를 통해 획득하는 결과값을 데이터 로드 파트에서 수행하여 파라메터로 넘겨줄 수 있도록 수정하였으며, GPU 내에 전송할 데이터 타입, 크기를 확인하고 최소화를 수행하였다. 가능한 경우 반복문 내부의 연산 구문들을 정리하여 최적화 수행하였으며, GPU 공유 메모리 사용여부에 따라 코드 변환 설계하였다. 결과 데이터 저장의 경우 GPU에서 CPU로 전송되는 데이터 크기 최소화하였다.

Ⅳ. GPU 기반 분산병렬처리 코드 변환 및 성능 테스트

HLLD의 GPU 활용을 위한 메모리 및 알고리즘 변환을 위해 입력 변수에서 bx, wl, wr의 경우 배열 변수이기 때문에 전체 복사해서 GPU에 전송할 경우 통신 비용이 크게 증가하므로, 필요한 값들만을 추출하여 새로운 메모리 구조에 저장하여 전송하는 것이 적합하다. 출력 변수의 경우 flx의 경우 for 반복문 내부에서 값을 저장하는 부분이 존재하며, 이를 해결하기 위해 빈 배열을 생성하여 GPU 내부의 결과 값을 저장하여 받아올 수 있도록 메모리 구조를 생성해야 한다. ey, ez, wct의 경우, GPU로부터 반환된 flx 값을 이용하여 계산하기 때문에 추가 메모리 구조를 구현할 필요가 없다. wct의 경우, 외부 함수를 호출하여 계산하는데, GPU 내부에서는 모든 외부 함수 호출이 어렵기 때문에 반드시 CPU에서 처리할 수 있도록 해야 한다. 함수 내 사용 변수인 peos의 경우 외부 호출함수들을 다수 포함하고 있기 때문에, GPU 내부에서는 활용이 어렵다. 그러므로 해당 변수를 통해 계산되는 for 반복문 내부의 변수들은 미리 계산하여 CPU에서 GPU로 전송해야하는 파라미터에 추가해야 한다. 반복문 이전에 계산되는 변수들은 미리 계산하여 GPU 커널 부분에 파라미터로 입력하는 것이 바람직하며 특히 igm1 같은 경우, 외부 함수를 호출하여 계산되기 때문에 반드시 GPU 커널

호출 전 작업이 필요하다. 그림 3은 입력 변수를 위한 GPU 메모리 할당과 그림 4는 반복문 병렬처리를 위한 커널 호출, 동기화, 결과 수신, 그림 5은 최종 결과 처리 및 반환을 보여주고 있다.

```
// Allocate GPU memory
Real *d_bx, *d_wl, *d_wr, *d_flx;
Real *peos_ule, *peos_ure, *peos_cfl, *peos_cfr;

cudaMalloc((void**)&d_bx, nx * d_NWAVE * sizeof(Real));
cudaMalloc((void**)&d_wl, nx * d_NWAVE * sizeof(Real));
cudaMalloc((void**)&d_wr, nx * d_NWAVE * sizeof(Real));
cudaMalloc((void**)&d_flx, nx * d_NWAVE * sizeof(Real));

cudaMalloc((void**)&peos_ule, nx * sizeof(Real));
cudaMalloc((void**)&peos_ure, nx * sizeof(Real));
cudaMalloc((void**)&peos_cfl, nx * sizeof(Real));
cudaMalloc((void**)&peos_cfr, nx * sizeof(Real));

// Copy data to GPU : tmpwli, tmpwri, tmpbxi to wli, wri, bxi
cudaMemcpy(d_bx, tmpbxi, nx * d_NWAVE * sizeof(Real), cudaMemcpyHostToDevice);
cudaMemcpy(d_wl, tmpwli, nx * d_NWAVE * sizeof(Real), cudaMemcpyHostToDevice);
cudaMemcpy(d_wr, tmpwri, nx * d_NWAVE * sizeof(Real), cudaMemcpyHostToDevice);

cudaMemcpy(peos_ule, pULE, nx * sizeof(Real), cudaMemcpyHostToDevice);
cudaMemcpy(peos_ure, pURE, nx * sizeof(Real), cudaMemcpyHostToDevice);
cudaMemcpy(peos_cfl, pCFL, nx * sizeof(Real), cudaMemcpyHostToDevice);
cudaMemcpy(peos_cfr, pCFR, nx * sizeof(Real), cudaMemcpyHostToDevice);
```

그림 3 입력 변수를 위한 GPU 메모리 할당

```
// Launch the kernel with GPU architecture flags
RiemannSolverKernel<<<blocksPerGrid, threadsPerBlock>>>(il, iu, ivx, ivy, ivz, d_bx, d_wl, d_wr, d_flx, nx, peos_ule, peos_ure, peos_cfl, peos_cfr, d_NWAVE, igm1);

// Ensure synchronization for debugging and correctness
cudaDeviceSynchronize();

// Copy results back to CPU
cudaMemcpy(tmpflxi, d_flx, nx * d_NWAVE * sizeof(Real), cudaMemcpyDeviceToHost);
```

그림 4 반복문 병렬처리를 위한 커널 호출, 동기화 및 결과 수신

```
//copy results to flx, ey, and ez with k, j, i
//call GetWeightForCT() for wct(k,j,i)
for (int i=il; i<=iu; ++i) {
  int tmpidx2 = i - il;
  int idx2 = tmpidx2 * d_NWAVE;

  flx(IDN,k,j,i) = tmpflxi[idx2 + IDN];
  flx(ivx,k,j,i) = tmpflxi[idx2 + IVX];
  flx(ivy,k,j,i) = tmpflxi[idx2 + IVY];
  flx(ivz,k,j,i) = tmpflxi[idx2 + IVZ];
  flx(IEN,k,j,i) = tmpflxi[idx2 + IEN];
  ey(k,j,i) = -tmpflxi[idx2 + IBY];
  ez(k,j,i) =  tmpflxi[idx2 + IBZ];

  wct(k,j,i) = GetWeightForCT(tmpflxi[idx2 + IDN], wl(IDN,i), wr(IDN,i), dxw(i), dt);
}
```

그림 5 최종 결과 처리 및 반환

```
__global__ void RiemannSolverKernel
(const int il, const int iu, const int ivx, const int ivy, const int ivz,
 const Real *bx, const Real *wl, const Real *wr, Real *flx,
 const int nx, const Real *peos_ule, const Real *peos_ure, const Real *peos_cfl,
 const Real *peos_cfr, const int dNWAVE, const Real igm1)
```

그림 6 반복문 처리를 위한 GPU Kernal

그림 6에서 보여주는 것처럼 반복문 처리를 위한 GPU Kernal에서 il, iu는 배열 내 접근 시작값 및 종료값, bx, wl, wr는 데이터 배열, flx는 결과 데이터 저장을 위한 데이터 배열, nx는 배열 접근 범위, peos_ule, peos_ure, peos_cfl, peos_cfr는 기존 peos를 통해 계산되었던 변수값, dNWAVE는 GPU 커널 내에서 처리되는 배열의 크기를 설정하는 전역변수 값이다.

본 논문에서 제안한 방법의 성능 평가를 위해 Google Colab CPU(single core), A100 GPU(grid block size = 256) 환경에서 Blast 시뮬레이션을 이용하여 average time for cycle (sec/cycle)과 total number of cycles을 측정하였다. 표 6은 성능 평가 결과를 보여주고 있다. 제안 방법을 이용하였을 때, average time for cycle 측면에서 약 25배 성능 향상되었으며 total number of cycles 측면에서 12.6% 성능 저하된 이유는 hlld.cu 변환 과정에서 max, min 함수를 외부의 std 라이브러리에서 호출하지 않고, 내부에서 구현하는 과정에서 결과가 일부 다르게 나와 추가적인 계산이 요구되었기 때문이다. 최종적 으로 단일 cycle 측면에서 성능은 크게 향상되었으나 전체 수행 시간은 약 20%밖에 향상되지 않았다. 이는 GPU에의 데이터 전송 및 결과 수신 과정에서 통신 비용이 추가되었기 때문이며, 이를 해결하기 위해서는 GPU 특성에 맞춘 메모리 최적화가 추가 연구로 필요하다.

표 9. 성능 평가 결과

	Sec/Cycle	Total # of cycles
A100 GPU	0.022732377	366
CPU	0.502123875	325

V. 결 론

본 논문은 HPC 환경에서 복잡한 계산을 요구하는 천체물리학 시뮬레이션 Athena++의 성능 최적화 방안을 도출하여 본질적으로 병렬화가 필요한 HPC 응용 연산에 대한 가속 전략을 제안하였다. 우선 코드 프로파일링을 통해 주요 병목 지점이 Hydro::RiemannSolver임을 정밀하게 식별하였으며 이를 해결하기 위해 HLLD solver의 핵심 연산을 CUDA 커널로 구현하여 CPU-GPU 이기종 시스템에서의 가속 전략을 제안하였다. 성능 실험은 Google Colab 환경에서 A100 GPU와 단일 CPU 코어를 기준으로 수행되었으며, 단일 연산 사이클 기준 평균 실행 시간이 약 25배 향상되는 성과를 보였다. 그러나 전체 실행 시간에서는 약 20% 향상에 그쳐, 병렬화만으로는 충분한 성능 개선을 달성하기 어려우며 GPU-CPU 간 데이터 이동 및 동기화 오버헤드가 전체 성능에 중대한 영향을 미침을 확인할 수 있었다. 이러한 결과는 HPC 환경에서의 코드 병렬화가 단지 연산 커널의 GPU 이전만으로 충분하지 않으며, 전체 시스템 관점에서의 병렬화 전략과 데이터 흐름 최적화가 병행되어야 함을 시사한다. 또한, 차세대 디바이스 (CXL(Compute Express Link), DPU(Data Process Unit) 등)를 포함한 이기종 아키텍처에 최적화된 메모리 구조 설계와 효율적인 스케줄링 정책이 향후 연구에서 중요한 역할을 할 것으로 판단된다.

결론적으로 본 논문은 실질적인 수치해석 코드에 병렬화 기술을 적용하고 성능을 실험적으로 검증함으로써, HPC 응용 분야에서 GPU 병렬 처리의 효과성과 한계를 동시에 조명하였다. 한계 극복을 위하여 본 논문에서 제시된 분석 방법과 최적화 전략을 다양한 차세대 디바이스 기반의 이기종 HPC 아키텍처에 적용하는 연구가 필요하다. 이는 다양한 과학·공학 시뮬레이션 코드의 병렬화 설계에 있어 실질적이고 유용한 기반을 제공할 수 있을 것으로 기대된다.

REFERENCES

[1] TIAN, Hongzheng, et al. HeteroBench: Multi-kernel Benchmarks for Heterogeneous Systems. *In: Proceedings of the 16th ACM/SPEC International Conference on Performance Engineering*. 2025. pp. 320-333.

[2] CHOI, Jiheon, et al. When HPC Scheduling Meets Active Learning: Maximizing The Performance with Minimal Data. *In: Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*. 2025. pp.

99-109.
[3] DEVARAJAN, Hariharan, et al. DFTracer: An Analysis-Friendly Data Flow Tracer for AI-Driven Workflows. *In: SC24: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE*, 2024. pp. 1-24.
[4] LOKHANDE, Mukul; RAUT, Gopal; VISHVAKARMA, Santosh Kumar. Flex-PE: Flexible and SIMD Multiprecision Processing Element for AI Workloads. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2025.
[5] EBRAHIMI POUR, Neda; DRESSEL, Frank; ROLLER, Sabine. Towards an Automated AI Based Simulation Framework in Aerospace Engineering. *In: Proceedings of the 2025 International Conference on High Performance Computing in Asia-Pacific Region Workshops*. pp. 58-60, 2025.
[6] Seungwoo Lee , Youn Doh Ha,"MPI-OpenMP Hybrid Parallelization for Multibody Peridynamic Simulations," *J.Comput. Struct. Eng. Inst.* vol. 33, no. 3, pp. 171-178, 2020.
[7] Chayanon Namo Wichitrnithed; Woo-Sun Yang; Yun Helen He; Brad Richardson; Koichi Sakaguchi; Manuel Arenaz,"Optimizing the Weather Research and Forecasting Model with OpenMP Offload and Codee," *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis(IEEE)*, 08 Jan., 2025.
[8] Myungho Lee, "Comparative and Combined Performance Studies of OpenMP and MPI Codes," *KTCCS*, vol. 13, no. 2, pp. 157-162, 2006.
[9] NASAR, Abouzied, et al. Task-parallelism in SWIFT for heterogeneous compute architectures. *arXiv* preprint arXiv:2505.14538, 2025.
[10] Amadou Chaibou and Oumarou Sie, "Comparative Study of the Parallelization of the Smith-Waterman Algorithm on OpenMP and Cuda C," *Journal of Computer and Communications* vol. 3, no. 6, pp. 107-117, 2015.
[11] GONG, Munan, et al. Implementation of chemistry in the Athena++ code. *The Astrophysical Journal Supplement Series*, 2023, 268.2: 42.
[12] CHAN, Yan-Mong, et al. Particle clustering in turbulence: Prediction of spatial and statistical properties with deep learning. *The Astrophysical Journal*, 2023, 960.1: 19.
[13] WHITE, Christopher J.; STONE, James M.; GAMMIE, Charles F. An extension of the Athena++ code framework for GRMHD based on advanced Riemann solvers and staggered-mesh constrained transport. *The Astrophysical Journal Supplement Series*, 2016, 225.2: 22.

저 자 소 개

정현미(정회원)
2010년 한남대학교 컴퓨터공학과 석사 졸업
2014년 한남대학교 컴퓨터공학과 학과 박사 졸업
2012년~현재 한국과학기술연구원 슈퍼컴퓨팅기술개발센터 선임연구원
<주관심분야 : HPC, 이기종컴퓨팅 , 병렬컴퓨팅, 클라우드 >

이현조(정회원)
2008년 전북대학교 컴퓨터공학과 석사 졸업
2014년 전북대학교 컴퓨터 공학과 박사 졸업
2015년~2016년 한국과학기술정보연구원 선임연구원
2017년~현재 : 한국농수산대학교 연구원
<주관심분야 : 병렬 질의처리, 암호화 빅데이터, AI >

정기문(정회원)
2001년 전남대학교 전산통계학 석사 졸업
2009년 전남대학교 정보보호학 박사 졸업
2001~2004년 한국정보보호진흥원 연구원
2004~2005년 국가정보원 연구원
2005년~현재 한국과학기술정보연구원 슈퍼컴퓨팅기술개발센터 책임연구원
<주관심분야 : HPC 클라우드, 이기종 컴퓨팅, 클라우드 보안 >

채철주(종신회원)
2009년 한남대학교 컴퓨터공학과 박사 졸업
2009년~2013년 한국전자통신연구원 선임연구원
2013년~2016년 한국과학기술정보연구원 선임연구원
2016년~현재 한국농수산대학교 교양학부 교수
<주관심분야 : 빅데이터, 인공지능, 디지털농업>