ISSN 2383-630X(Print) / ISSN 2383-6296(Online)
Journal of KIISE, Vol. 46, No. 10, pp. 975-980, 2019. 10
https://doi.org/10.5626/JOK.2019.46.10.975

CPU-GPU 컨테이너 클러스터의 프로파일링을 활용한 계산응용 실행 계획 기법

(An Execution Planning Scheme for Computational Applications using Profiling of a CPU-GPU Container Cluster)

오 지 선 † 김 세 진 †† 김 윤 희 †††
(Jisun Oh) (Sejin Kim) (Yoonhee Kim)

요 약 이기종 자원을 활용한 클러스터 및 클라우드 환경이 보편화되면서 자원의 활용도를 높이기 위한 자원 선택 환경이 필요하다. 응용의 실행 특성에 따라 적절한 자원 사용 패턴이 다를 수 있으므로 자원 사용에 따른 적절한 실행 계획이 필요하다. 본 논문에서는 수집한 응용 실행의 프로파일링 정보를 이용하여 CPU-GPU 컨테이너 클러스터 환경에서 응용별 GPU 노드 자원 배치율을 계산한다. 이를 통해 GPU 노드에 적합한 응용을 선택하고, 배치율이 낮은 응용은 CPU 노드에 배치함으로써 자원을 적절하게 분배한다. 또한 응용의 프로파일링 정보를 이용하여 GPU 메모리 사용을 예측하여 GPU 노드에서 실행 중인 작업의 실행 순서를 계획함으로써 수행시간이 단축할 수 있음을 증명하였다.

키워드: 컨테이너, 응용 프로파일링 데이터, 응용 할당, 작업 실행 계획, 쿠버네티스

Abstract As heterogeneous clusters and cloud environments have gained popularity, selection of appropriate resources in an integrated environment is considered essential for enhancement in application performance with reasonable resource utilization. Application characteristics may require specific resource in a certain order during its execution and ask smart job deployment among diverse nodes. Especially, it is necessary to have an execution plan in advance for better performance of CPU and GPU container clusters. In this paper, we propose an execution planning scheme based on runtime profiling history to place jobs on CPU-GPU nodes. Computational applications usually show good performance in GPU. However, the lack of GPU sharing methods leads to failure of co-locating jobs on a GPU node. Based on the profile information, the scheme provides the combination of applications to run at the same time in GPU container clusters and estimate the performance of workload before executing application among CPU-GPU container clusters. We have also demonstrated adjustment of the order of application execution using profile history in order to reduce the execution time of total workload by monitoring GPU memory usage.

Keywords: container, application profiling data, application assignment, task execution planning, kubernetes

· 이 성과는 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(No. NRF-2017R1A2B400568, 2015M3C4A7065646)
· 이 논문은 2018 한국소프트웨어종합학술대회에서 '연산 응용 특성에 따른 CPU-GPU 자원 조절 클라우드 실행 환경 설계'의 제목으로 발표된 논문을 확장한 것임

† 학생회원 : 숙명여자대학교 컴퓨터과학과
jsoh8088@gmail.com
†† 학생회원 : 숙명여자대학교 소프트웨어학부
wonder960702@gmail.com
††† 종신회원 : 숙명여자대학교 소프트웨어학부 교수
(Sookmyung Women's Univ.)
yulan@sookmyung.ac.kr
(Corresponding author임)

논문접수 : 2019년 3월 4일
(Received 4 March 2019)
논문수정 : 2019년 5월 14일
(Revised 14 May 2019)
심사완료 : 2019년 6월 10일
(Accepted 10 June 2019)

정보과학회논문지 제46권 제10호(2019. 10)

1. 서 론

하드웨어 기술의 발전에 따라 최근 대량의 프로세싱 코어를 제공하는 GPU(Graphics Processing Unit)의 높은 병렬 처리 연산으로 범용 CPU와 함께 사용되면서 높은 컴퓨팅 성능을 구현한다. GPU의 빠른 연산 속도의 장점으로 인해 가상화된 클라우드 컴퓨팅 환경에서도 GPU 자원의 제공이 널리 보급되고 있다. 클라우드 환경에서 한정적인 자원과 사용자의 요구사항 기반하에 GPU의 활용도를 향상시키기 위해 GPU에 대한 이해와 여러 응용 프로그램의 GPU 자원의 효율적인 공유에 대한 연구가 필요하다.

가상화 기술인 VM(Virtual Machine)을 기반으로 한 클라우드 컴퓨팅, 대규모 분산 컴퓨팅 환경과 같은 플랫폼은 배포 및 파기 등 관리로 인한 오버헤드가 발생한다. 이러한 성능 오버헤드 문제를 해결하기 위해 컨테이너 기술은 전체 가상 컴퓨터를 실행하지 않고 분산된 응용 프로그램을 배포하고 작동함으로써 단순화시킨다. 그러나 컨테이너의 장점에도 불구하고 컨테이너의 자원 경합 문제로 인한 성능 오버헤드는 발생한다.

AWS ECS[1]의 경우, CPU 또는 메모리 최소량에 따른 작업 배치 전략을 통해 자원의 수를 최소화한다. 실행 특성이 다양한 응용을 수행할 때, 제한된 GPU 자원을 활용하는 실행 환경을 구축할 경우, 선택적 GPU 자원을 사용을 계획하는 것이 필요하다. 가상화된 CPU-GPU 컨테이너 환경에서 응용 프로파일링 정보, 사용자의 요구사항을 반영하여 컨테이너의 자원 공유 및 작업의 적절한 자원 배치를 해야 한다. 응용 프로그램의 실행 패턴을 파악하여 적용하는 것이 필요하다.

본 논문에서는 컨테이너 별 자원 공유를 위해 수집한 응용의 프로파일링 정보를 이용하여 응용 별 GPU 자원 배치율을 계산하고 결과에 따라 각 자원에 배치한다. 또한 프로파일링 정보를 통해 GPU 자원에 배치된 실행 중인 응용의 자원 사용을 예측하여 자원 부족이 발생하지 않고 수행시간이 단축할 수 있는 실행 계획을 수립한다. 실험에서 다양한 계산응용을 대상으로 초기 작업 배치와 GPU 배치되는 작업의 실행순서에 따른 실행시간을 비교하여 적절한 배치가 전체 실행시간을 감소시킴을 보여준다.

본 논문의 구성은 다음과 같다. 1장의 서론에 이어 2장에는 관련 연구를 살펴본다. 3장에서는 컨테이너 기반 CPU-GPU 자원 및 작업 실행 계획 모델을 정의한다. 4장에서는 CPU-GPU 자원 및 작업 실행 계획 전략을 보이며, 5장에서는 실험을, 6장에서는 결론을 맺는다.

2. 관련 연구

클라우드 또는 클러스터 환경에서 GPU 자원 경쟁 해결 및 CPU-GPU 작업 분배에 관련된 연구가 있었다. Steran[2]의 경우 컨테이너를 통해 CPU와 GPU 메모리 간의 데이터 재배포 메커니즘을 보여준다. 그러나 기계학습 응용의 CPU 메모리와 GPU 메모리 간의 재배포는 오버헤드를 발생시킨다. 또한 Piereira[3]의 연구는 CPU-GPU 시스템 환경에서 데이터와 계산을 분할하여 CPU와 GPU에 할당하는 프레임워크를 제안하였다. 하지만 [3]의 연구에서는 CPU 대 GPU의 작업 분할에 대한 적절한 판단 기준을 고려하지 않았다. Theodora[4]의 연구는 컨테이너 환경에서 프로파일링 데이터를 기반으로 하여 CPU 메모리를 동적으로 할당하는 기법을 제안하였다. 자원의 유휴 상태를 확인하여 컨테이너에 할당하지만 미리 자원의 유휴 상태를 예측하여 할당하지 않는다.

NVIDIA는 각 컨테이너에 개별적으로 설치하지 않고도 호스트에서 컨테이너로 GPU 드라이버를 쉽게 공유 및 가상화하는 방법인 NVIDIA Docker를 제시하였다. NVIDIA Docker가 물리적 자원을 할당하는 방식은 일반적으로 단일 컨테이너로 제한된다.

그러나 GPU를 위한 컨테이너 기반 가상화 환경에서 발생할 수 있는 문제를 고려해야 한다. 컨테이너 별 GPU 자원 공유는 GPU 메모리 공유 기반 하에 이루어진다[5]. 이때 GPU 메모리가 부족하여 컨테이너에 할당되지 못하면 OOM(Out of memory)이 발생한다. 하나의 GPU 컨테이너가 존재하고 일정량의 GPU와 메모리를 사용하고 있다고 가정한다. 이때 두 번째 컨테이너가 생성되면 컨테이너 2에 GPU 메모리 할당을 시도한다. 그러나 GPU 메모리 공간의 크기가 부족하여 두 번째 컨테이너에 필요한 용량을 할당할 수 없으므로 OOM을 반환 한다. 결국 시스템은 메모리 크기를 제어하거나 메모리를 공유하는 대신 응용 실행을 취소한다. 위 문제와 같이 시스템이 제한된 자원을 활용하기 위해 응용의 자원 사용 패턴 및 자원의 적합성에 따라 자원의 분리가 필요하다. 실제로 필요한 자원의 양과 응용의 흐름에 따라 컨테이너에 GPU 메모리를 공유하고 제어할 수 있는 적절한 전략이 필요하다.

3. 컨테이너 기반 CPU-GPU 자원 및 작업 실행 계획 모델 정의

전체적인 시스템 디자인은 그림 1과 같다. 전체 시스템은 프레임워크(framework), 커널(Kernel) 및 인프라스트럭처(Infrastructure), 3가지 레이어로 분류된다. 프레임워크는 응용 프로그램 프로파일링 데이터, 자원 컨트롤 매니저, 스케줄러 및 모니터링 모듈을 포함한다. 본 논문에서 제안하는 알고리즘은 이 레이어 스케줄링 모듈에 포함되어 있다.

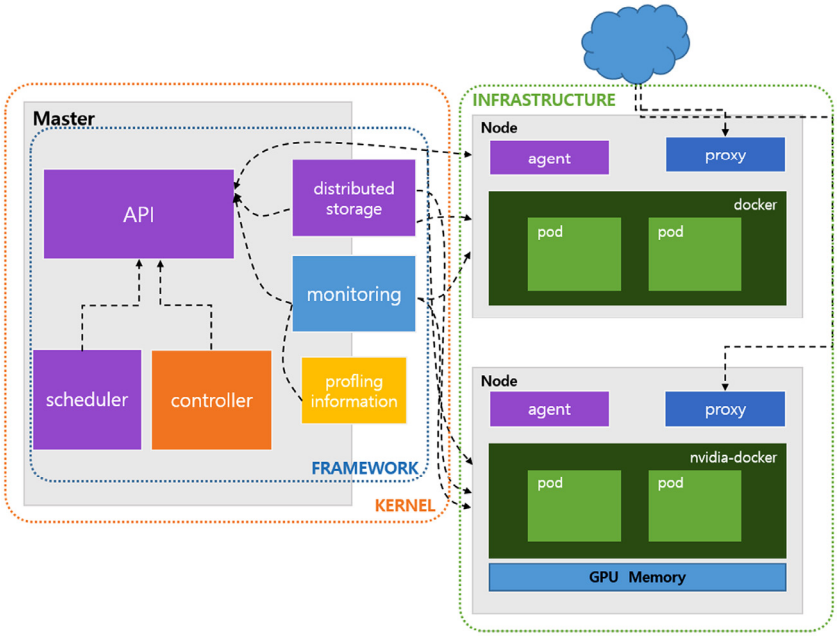


그림 1 컨테이너 기반 클러스터 구조도
Fig. 1 Container-based Cluster Architecture

자원 관리 매니저는 리소스 관련 작업을 생성하고 리소스의 조정을 관리한다. 할당할 GPU 자원 개수, 컨테이너 수, 네트워크 설정, 환경 변수 설정을 설정하여 작업을 생성한다. 스케줄러는 일반적으로 각 애플리케이션의 프로파일링 된 정보를 확인한 후에 수행된다. 이 단계의 주요 역할 중 하나는 응용 프로그램의 자원 사용을 예측하고 GPU 컨테이너 노드에 작업을 적절히 배치하는 계획을 수립한다. 또한 컨테이너의 수행 순서 및 실행 중 플랜을 재구축할 수 있다. 응용 프로그램 프로파일링 모듈의 주요 역할은 응용 프로그램의 특성을 제공하고 분석하여 스케줄러에 제공한다.

두 번째 및 세 번째 레이어는 실제 작업을 수행하는 마스터 노드, 리소스 풀로 구성되어있다. 마스터 노드는 클러스터의 자원을 식별하고 할당 및 관리한다. 마스터 노드는 컨테이너 클러스터 프레임워크인 Kubernetes[6] master를 기반으로 한다. 시스템은 단일 컴퓨터에서 여러 노드의 자원을 제어하고 사용 가능한 자원 정보를 인식하여 자원을 동적으로 할당할 수 있다. 작업 스케줄러에 의해 생성된 스케줄에 따라 각 노드의 컨테이너의 리소스(CPU, GPU, 메모리, 디스크 등) 등을 pod 형태로 감지하고 제어한다. 인프라 스트럭처 레이어의 노드는 각각 docker[7], nvidia-docker[8] 기반 컨테이너로 구성되며 모든 노드에서 실행되며 실행 중인 pod를 유지관리하고 Kubernetes 런타임 환경을 제공한다. 컨테이너는 각 노드에서 관리되며, kubelet을 통해 컨테이너가 실행되는지 확인한다. Kube-proxy를 통해 마스터 노드에 네트워크 연결 및 전달한다.

4. CPU-GPU 자원 및 작업 실행 계획 전략

본 장에서는 컨테이너 기반 클러스터 환경에서 GPU 자원 점유 경쟁 문제를 해결하기 위해 프로파일링 정보를 기반으로 한 우선순위 기반 GPU 자원 고려 작업 할당 및 오버래핑을 통한 작업의 자원 배치 및 작업 실행 순서 계획 전략을 제안한다.

4.1 우선순위 기반 초기 작업 할당 전략

응용은 입력하는 파라미터에 따라 각 단계에서의 자원 사용량이 일정하게 변화한다. 그러나 응용의 해석하고자 하는 대상이 변하는 경우에는 자원 사용 패턴이 달라지게 된다[9]. 따라서 본 논문의 응용의 프로파일링 정보는 수행하고자 하는 대상이 같은 경우, 입력 데이터의 크기, 파라미터가 달라지는 경우의 조건에만 수집 가능하다. 또한 프로파일링을 통한 오버헤드는 발생하나 자원 부족으로 응용 실행 실패를 통한 재실행의 수를 줄일 수 있으므로 전반적인 워크로드의 수행시간을 감소시킬 수 있다.

우선순위 기반 초기 작업 할당 전략은 다음과 같다. 우선 전체 대기열의 응용 프로파일링 정보를 수집한다. 응용의 프로파일링 정보는 수행시간, GPU 메모리 사용

률 및 사용자가 지정한 step이다(식 (1)). 이를 기반으로 응용의 GPU 메모리와 프로파일링 정보의 가중치를 계산하여 GPU 노드 자원 배치율을 결정한다(식 (2)). GPU 메모리 할당 비율에 따라 컨테이너에 할당하며 이때 모니터링을 통해 대기열의 남은 작업과 GPU 메모리 사용률 파악하여 업데이트 한다.

AWS ECS의 작업 배치 전략에 따르면 CPU 또는 메모리 최소량 정보를 통해 자원의 수를 최소화하여 서비스 제공자 입장에서 사용 중인 자원의 수를 조절한다. 이를 GPU 자원에도 적용하여 GPU 메모리와 같이 제한된 자원의 활용도를 높일 수 있다. 응용의 GPU 메모리 사용량, 수행시간 등의 프로파일링 정보를 통해서 대기열 중 효율적인 GPU 자원 사용 응용의 우선순위를 정하고 우선순위 뒤에 있는 작업들은 CPU 노드에 배치하여 오버래핑할 수 있다. 이를 통해 GPU 자원의 완전한 활용뿐만 아니라 대기열의 총 수행시간을 줄일 수 있다.

$$prof_i = \begin{Bmatrix} job_i, CPUexecutiontime, \\ GPUexecutiontime, GPUmemutil, step_j \end{Bmatrix} \quad (1)$$

$$rate_i = C_0 * \frac{j_i}{\sum_{j=1}^{n} j_i} + C_1 * \frac{prof_i^1}{\sum_{j=1}^{n} prof_j^1} + \ldots + C_{m-1} * \frac{prof_i^{m-1}}{\sum_{j=1}^{n} prof_j^{m-1}} \quad (2)$$

4.2 GPU 작업 실행 순서 계획 전략

응용의 프로파일링 정보를 기반으로 GPU 작업 실행 순서 조절에 사용할 step을 정의한다. step은 각 응용 프로그램의 수행시간과 GPU 메모리 사용량을 기반으로 사용자가 정의한다.

작업 실행 순서 조절 전략은 다음과 같다. 각 실행중인 워크로드 그룹의 작업들의 프로파일링 정보인 step과 현재 실행 중인 step의 정보를 받아온다(line 1-7). 프로파일링 정보인 step이 현재 사용하고 있는 GPU 메모리 자원 사용량 보다 많다면 전체 GPU 메모리에 저장한다(line 8-10). 각 수행 중인 워크로드의 GPU 메모리 사용 예측 정보를 저장하고(line 12), 실제 GPU 자원의 메모리보다 GPU 메모리 사용 예측 정보가 클 경우, 각 작업들의 우선순위를 계산하고 우선순위가 낮은 GPU 컨테이너를 일시중지(pause)시킨다(line 13-15). 아닌 경우에는 GPU 컨테이너를 다시 실행(unpause)시켜 응용의 실행 순서를 제어한다.

5. 실험 및 결과

본 장에서는 앞에서 제안한 알고리즘을 평가하기 위해 실험을 진행하였다. 첫째, 우선순위 기반 초기 할당 전략을 통해 작업 별 컨테이너 자원 할당 결정 및 실행을 포함한다. 둘째, 실행 중인 컨테이너의 작업 실행 순서 조절 전략의 성능을 분석한다.

Algorithm 1 GPU Task Execution Planning Algorithm

```
1:  function TASKEXECUTIONPLANNING(G_j)
2:      STATE ← Running;
3:      while (Running) do
4:          If task_i in G_j is Running
5:              if step_i in task_i is Running
6:                  step_i ← Profiled memory requirement(mem_i + α);
7:                  Pmem_i ← Present used memory(step_{i-1});
8:                  if step_i ≥ Pmem_i then
9:                      GM[i] += step_i;
10:                 end if
11:             end if
12:             sumGM += GM[i];
13:             if GlobalGpuMem < sumGM then
14:                 CALCULRATE( task );
15:                 GC_i ← Pause;
16:             else
17:                 GC_i ← Unpause;
18:             end if
19:         NextInterval();
20:         end if
21:     end while
```

그림 2 GPU 작업 실행 순서 계획 알고리즘
Fig. 2 GPU Task Execution Planning Algorithm

5.1 실험 환경

실험에 사용된 시스템은 2개의 Intel i7 CPU, 32GB Ram 및 12GB 메모리가 장착된 NVIDIA GeForce Titan Xp GPU 4장으로 구성되어 있다. 본 실험 환경으로는 컨테이너 클러스터 플랫폼 Kubernetes을 사용하여 docker, nvidia-docker 컨테이너 기반 1개의 마스터 노드와 CPU 컴퓨트 노드, GPU 노드를 구성하였다. 실험에 사용된 컴퓨팅 머신은 아래 표 1과 같다.

표 1 실험 환경 정보
Table 1 Information on Experiment Environment

	CPU	GPU
Architecture	Intel(R) Core(TM) i7-5820K	Nvidia GeForce TItan Xp D5x
Core Clock	3.30GHz	1.58GHz
Num of Cores	6 cores	128 CUDA cores
Mem. size	32 GB	12 GB
Threading API	-	Nvidia CUDA 10.0
Compiler	ICC (Intel Compiler)	Nvidia C Compiler (NVCC8.0)
OS	Ubuntu 16.04.3 LTS	Ubuntu 16.04.3 LTS

5.2 대상 응용 및 응용 프로파일링 정보

CNN-MNIST는 주로 복합 영상 알고리즘으로 영상 이미지 분석에 주로 사용된다. CNN은 여러 개의 숨겨진 레이어로 구성되며 일반적으로 회선 레이어와 풀링 레이어로 구성되어 수행한다. 숫자 손글씨 이미지인

MNIST 데이터 세트를 사용하여 분석한다. Binomial 모델은 금융 상품의 시간에 따라 가격이 다른 이산 시간 모델을 사용하여 이산 시간의 근사를 평가하며, Black scholes 역시 역학 금융 시장의 수학적 모델로 편미분 방정식으로부터 가격에 대한 추정치를 제공한다. LAMMPS는 분자 동역학 시뮬레이션을 위해 설계된 응용 소프트웨어로 원자 모델링이나 원자, 메소산, 연속체 스케일에서 병렬입자 시뮬레이터로 사용된다. 이는 메시지 전달 기술과 시뮬레이션 도메인의 공간 분해를 사용하여 단일 프로세서 또는 병렬로 실행된다. 실험에서는 Leanard Jones의 3D melt 예제를 사용하였다. GROMACS는 단백질 및 지질을 시뮬레이션 하는데 사용되는 분자 동역학 응용 프로그램이다. 수백에서 수백만 개의 입자가 있는 시스템에서 뉴턴의 운동 방정식을 시뮬레이션 한다. 본 실험에서는 1536K의 물 분자 데이터 셋을 테스트하였다.

사용한 응용의 프로파일링 정보는 표 2와 같다. CNN-MNIST 응용은 CPU, GPU 노드에서 실행했을 경우, 각각 3752초, 324초 실행 시간이 소요되며 GPU 메모리 사용량은 1228MB이다. Black scholes 응용은 CPU, GPU 노드에서 실행했을 경우, 각각 324초, 21초 소요되며 GPU 메모리 사용량은 483MB이다. Binomial 응용의 경우 CPU, GPU 노드에서 실행한 결과 각각 295초, 32초 걸리며 GPU 메모리 사용량은 249MB이다. LAMMPS 응용의 경우 CPU, GPU 노드에서 각각 990초, 85초 수행되며 GPU 메모리 사용량은 300~2273MB까지 증가한다. Step 1은 GPU 메모리 사용량이 0~833MB, step 2는 GPU 메모리 사용량이 833~1895MB, step 3은 1895~2795MB, step 4는 2795~8317MB 이내인 경우로 정의하였다. GPU 메모리 증가하는 형태에 따라 step을 4로 지정하였다. 마지막으로 GROMACS 응용은 CPU, GPU 노드에서 각각 1251초, 287초 실행 시간이 소요되며, GPU 메모리 사용량은 371초이다.

본 실험을 위해 CNN, Black scholes, Binomial, LAMMPS, GROMACS 응용 각각 13개, 5개, 5개, 1개, 1개의 응용 총 25개의 작업을 구성하였다. 각 응용의 GPU 노드 배치율은 CNN 약 0.608, Black scholes 약 0.541, Binomial 약 0.533, LAMMPS 약 0.591, GROMACS 약 0.583이다. GPU 컨테이너 5개, CPU 컨테이너 5개에서 진행하였다.

표 2 응용 프로파일링 정보
Table 2 Application Profiling Information

	Execution Time(s) - CPU	Execution Time(s) - GPU	GPU Memory Usage(MB)	step
CNN-MNIST	3752	198	1228	1
Black Scholes	324	21	483	1
Binomial	295	32	249	1
LAMMPS	990	85	300~2273	4
GROMACS	1251	287	371	1

5.3 실행 결과

우선순위 기반 초기 작업 할당 전략에 따라 작업의 응용을 각 노드에 배치하여 실험을 진행하였다. 전략에 따라 계산된 배치율의 결과에 따라 CNN, LAMMPS 응용이 GPU 노드에 우선 배치되었으며, Black Scholes, Binomial, GROMACS 응용은 CPU 노드에 배치되어 실행하였다. 제안한 작업 할당 전략에 따라 CPU-GPU 노드에서 실험한 수행시간과 GPU 노드에 작업을 배치하여 실행한 수행시간을 비교하였다. GPU 노드에 배치하는 기본 배치의 경우에는 약 1693초가 소요되었으며, 우선순위가 낮은 작업을 CPU 노드에 배치한 경우에는 약 1294초 소요되었다. 초기 작업 할당 전략에 따라 실행하였을 때, 총 작업의 수행시간이 감소한 것을 알 수 있다.

GPU 노드에 배치된 작업의 실행 순서 계획 전략을 적용하여 GPU 메모리 OOM 발생 횟수와 수행 시간을 분석하여 성능을 비교하였다. 표 2의 응용의 프로파일링 정보의 사용자가 지정한 step을 통해 실행 중인 컨테이너의 2번의 일시 중지를 실행하였다.

그림 3은 GPU에 배치된 작업을 제안하는 전략을 적용하지 않고 실험한 결과이다. OOM은 약 745초, 1170초에 2번 발생하였으며, 수행시간은 1687초 소요되었다. 실행 중인 응용의 GPU 메모리 사용량을 알지 못하기 때문에 응용이 동시에 실행됨으로써 12GB GPU 메모리보다 더 높은 메모리를 요구하여 OOM이 발생하였다. 또한 OOM 발생으로 인하여 실패한 응용이 재실행됨으로써 수행시간이 늘어남을 알 수 있다.

그림 4는 작업의 실행 순서 조절 전략을 적용한 결과이다. 실험에서 LAMMPS 응용의 step 2, 3에서 다음 step으로 넘어가기 전 일시 정지를 통해 컨테이너 실행을

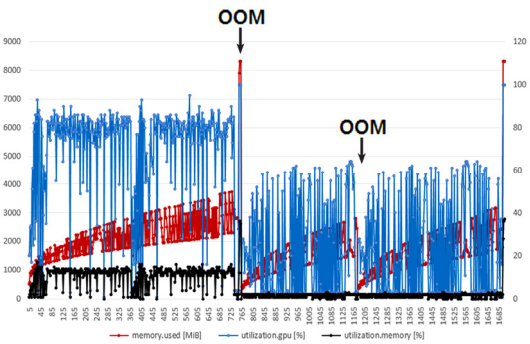


그림 3 기본 GPU 작업 실행 실험
Fig. 3 Default GPU Task Execution Experiment

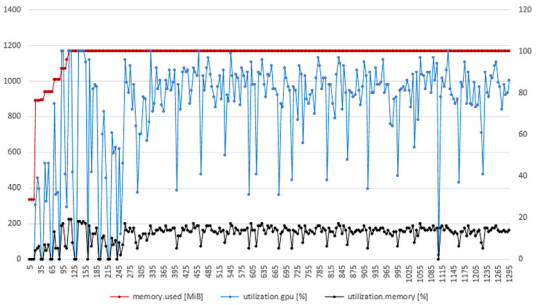


그림 4 GPU 작업의 실행 순서 계획 전략 실험
Fig. 4 GPU Task Execution Planning Experiment

제어하였다. 실행 후 약 730초, 1145초에 일시 중지하였다. OOM이 발생하지 않았으며, 수행 시간은 1294초이다. 실행되고 있는 응용의 다음 GPU 메모리 사용량을 예측하여 컨테이너를 중지하고 다시 실행함으로써 OOM이 발생하지 않는다. 또한 OOM을 통한 응용의 실패가 없어 응용의 재실행이 발생하지 않으므로 짧은 수행시간이 소요되는 것으로 분석된다.

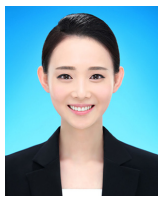
6. 결 론

본 논문에서는 응용의 프로파일링 정보를 이용하여 컨테이너 클러스터 환경에서의 작업의 초기 할당 배치 및 작업의 실행 순서 조절 전략을 제안하였다. 응용의 CPU에서의 수행시간, GPU에서의 수행시간 및 OOM이 발생할 수 있는 문제를 해결하기 위해 GPU 메모리 사용량과 step을 수집하였다. 수집한 프로파일링 정보를 통해 본 연구에서 정의한 초기 할당 배치율을 계산하여 GPU 노드에서 실행할 응용의 우선순위를 계산한다. 우선순위가 높은 응용을 GPU 노드에 우선 할당한 후 나머지 응용을 CPU 노드에 할당한다. 실험을 통해 전체 작업 실행 시 수행시간이 감소함을 보였다. 또한 응용 프로파일링 정보를 이용하여 GPU 메모리 사용량을 예측하여 실행 중인 GPU 컨테이너 노드의 실행 순서를 조절하였다. 제안한 전략을 적용하여 실험한 결과 응용의 OOM이 발생하지 않았으며, 수행시간이 감소함을 확인하였다.

향후 연구로는 컨테이너 및 여러 자원 클러스터 환경을 구축하고 I/O, 메모리, 네트워크 등 응용의 자원 사용 패턴을 다양하게 수집하여 적용하고자 한다. 또한 컨테이너 자원이 아닌 여러 자원의 특성 및 상태에 따른 적합한 자원 배치 기법으로 확장하고, 자원 사용 패턴을 이용하여 작업 실행 순서 조절의 일반화를 진행하고자 한다.

References

[1] Amazon ECS, [Online]. Available: https://aws.amazon.com/ko/ecs/
[2] Breuer, Stefan, et al., "Extending the SkelCL skeleton library for stencil computations on multi-GPU systems," *Proc. of the 1st International Workshop on High-Performance Stencil Computation*, pp. 15-21, 2014.
[3] Pereira, Alyson D., Luiz Ramos, and Luís FW Góes, "PSkel: A stencil programming framework for CPUGPU systems," *Concurrency and Computation: Practice and Experience*, Vol. 27, No. 17, pp. 4938-4953, Apr. 2015.
[4] Adufu Theodora, Jieun Choi, and Yoonhee Kim, "Dynamic Memory Allocation for Scientific Workflows in Containers," *Journal of KIISE : Computer Systems and Theory*, Vol. 44, No. 5, pp. 439-448, May. 2017.
[5] Kang, Daeyoun, et al., "ConVGPU: GPU management middleware in container based virtualized environment," *Proc. 2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 301-309, 2017.
[6] Kubernetes, [Online]. Available: https://kubernetes.io/
[7] docker, [Online]. Available: https://www.docker.com/
[8] nvidia-docker, [Online]. Available: https://github.com/NVIDIA/nvidia-docker
[9] Rhu, Minsoo, et al., "vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design," *Proc. The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, p. 18, 2016.



오 지 선
2017년 숙명여자대학교 컴퓨터과학과 졸업(학사). 2017년~현재 숙명여자대학교 컴퓨터과학과 석사과정. 관심분야는 클라우드 컴퓨팅, 분산컴퓨팅, GPU 가상화

김 세 진
정보과학회논문지
제 46 권 제 9 호 참조

김 윤 희
정보과학회논문지
제 46 권 제 9 호 참조