[2025-4]

KEASE 기술서

한국형 엑사스케일 응용 SW 개발 환경 프레임워크

슈퍼컴퓨터 5호기의 메모리 성능 최적화 및
가속기 상에서의 응용 프로그램 성능 최적화

2025. 03. 12

명 지 대 학 교

[개 정 이 력]

개정 일자	버전	설명	개정페이지	작성자	확인자
2025.3.12	1.0	INITIAL DRAFT	전체	이명호	이명호

목 차

1. 연구의 개요

최근의 슈퍼컴 아키텍쳐는 엑사스케일의 높은 성능 요구사항과 그에 비해 상대적으로 낮은 전력소모 요구사항(~20 Mwatt)을 동시에 충족하기 위해 범용 멀티코어 CPU 및 매니코어 GPU 등과 같은 가속칩들을 혼용한 이기종 아키텍쳐로 구축되고 있다.

가속칩의 아키텍쳐는 점점 더 복잡해지고 있고, 멀티코어 CPU에도 AVX와 같은 벡터연산 지원기능 등이 탑재되면서 복잡도가 크게 증가하고 있으며, 이러한 이기종 슈퍼컴 상에서 HPC 응용 프로그램을 위한 높은 성능을 얻기 위해서는 멀티코어 CPU와 매니코어 가속칩의 잠재적인 성능을 최대한 끌어낼 수 있는 병렬화 및 성능 최적화가 필수적이다.

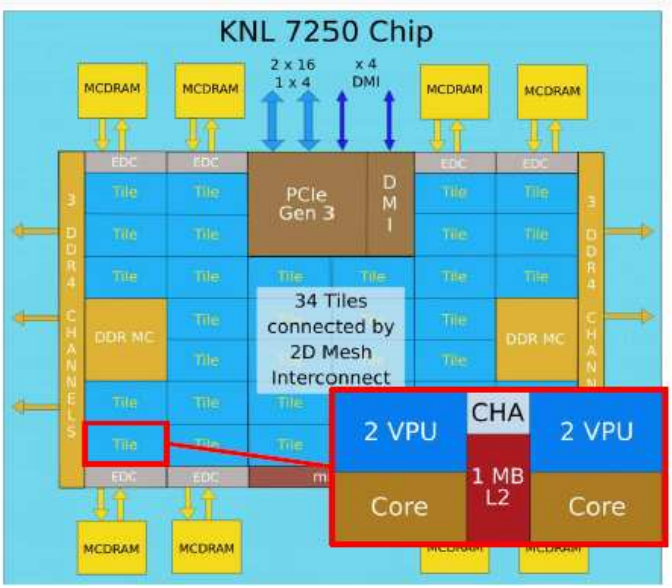
이를 위하여 슈퍼컴 5호기 Intel KNL 아키텍쳐 및 응용의 성능적인 특성에 대한 모델링과 이들에 기반하여 병렬화, 최적화 기술들을 개발하는 것이 중요하다. 2차년도의 연구에서는 1차년도의 연구 결과를 활용하여 Intel KNL의 메모리 성능 최적화 연구를 수행하였다. NUMA (Non-Uniform Memory Access) 특성을 갖는 KNL의 메모리 아키텍쳐에서 메모리 연산이 집중된 응용 프로그램은 사용하는 데이터의 home node 위치에 따라 벤치마크의 성능이 큰 폭으로 달라질 수 있다. 따라서 메모리 대역폭을 최적화하기 위하여 KNL의 NUMA에 friendly한 스레드 배치가 중요한데, 연구에서는 스레드 배치 방법에 따른 STREAM 벤치마크의 성능을 분석하여 최적의 배치법을 도출하였다. 또한 각 세부 벤치마크별로 성능을 세밀히 분석하여 메모리 연산과 산술 연산의 비중에 따른 성능의 추세를 분석하였다. 다음으로 MCDRAM의 활용이 KNL의 성능에 미치는 영향을 axpy, SOR, matrix add, dot product 등의 마이크로 벤치마크를 사용하여 분석하였다.

또한 가속기(GPU) 상에서의 응용 프로그램의 성능 최적화 연구도 수행하였는데, 먼저 수치 라이브러리 등 슈퍼컴 응용의 주요 커널 중 하나인 Cholesky decomposition의 성능을 최적화하였다. 이를 위하여 kokkos를 활용하였는데, 다차원 배열에 대한 최적의 메모리 할당 및 관리와 데이터 통신 최적화를 위하여 Kokkos::View 기능을 활용하였다. 그리고 GPU를 활용한 웹 기반의 이미지 프로세싱 응용, 웹 기반의 행렬 곱셈 알고리즘의 성능 최적화 연구도 수행하였다. 최근 HPC 플랫폼 상에서 활용도가 높아진 인공지능 응용을 타켓으로 한 성능 최적화 연구도 함께 수행하였다.

2. 슈퍼컴퓨터 5호기에서의 메모리 성능 최적화

1) 슈퍼컴 5호기의 Intel KNL 노드 상에서 스레드 배치를 통한 메모리 대역폭 최적화 연구

- Intel KNL (Knight Landing)은 64~72개의 silvermont 코어들로 이루어진 MIC (Many Integrated Core) 칩이다. 각 코어는 1.3~1.5GHz로 작동하며, 2개의 코어들은 하나의 tile을 구성한다 ([그림 1] 참조). 펌웨어 세팅을 통해 코어들의 클러스터링 구조를 변경하는 것이 가능한데, All-to-All, Hemisphere, Quadrant, SNC-2, SNC-4 모드들이 있다. 각 코어는 Hyper-Threading 방식으로 실행되는 4개의 hardware thread들을 지원한다. 코어당 2개의 512-bit VPU가 있는데, 동시에 8개의 double precision 연산들을 처리한다.
- KNL의 각 tile의 두 개의 코어들은 1MB 크기의 L2 캐시를 공유하는데, 데이터 공유를 위한 인터커넥트는 2-D Mesh를 사용한다. 16G~32Gbyte 크기의 MCDRAM (MultiChannel DRAM)과 384GB의 DDR DRAM을 사용하는데, MCDRAM의 대역폭은 450GB/sec, DDR DRAM은 90GB/sec이다. 펌웨어 세팅을 통해 MCDRAM을 캐시 방식으로 사용할지, 일반 메모리로 사용할지, 또는 캐시와 일반 메모리의 중간인 hybrid 모드로 사용할지 결정할 수 있다.

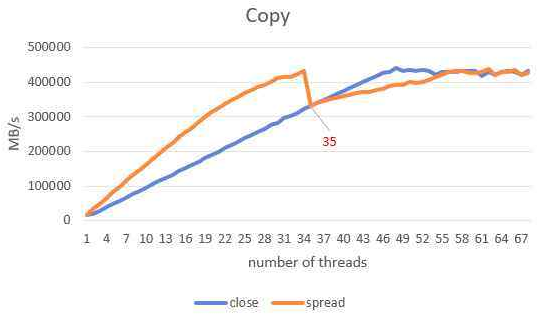


[그림 1] Intel KNL 아키텍처

- Intel KNL 상에서 메모리 연산이 집중된 응용 프로그램을 병렬 실행할 경우 여러 스레드/프로세스들로부터의 메모리 접근 연산들로 인해 캐시, 캐시 인터커넥트, 메모리 버스, 등에 걸리는 부하가 크게 높아져 메모리 대역폭에 병목현상이 발생하는 등 성능이 저하될 수 있는 바, 응용 프로그램의 성능 최적화를 위해서는 메모리 대역폭을 효율적으로 사용하는 것이 중요하다.
- STREAM 벤치마크를 사용한 실험을 통해 스레드의 배치 방식이 KNL의 성능에 미치는 영향에 대해 분석한다. STREAM은 4가지 연산들 Copy (c[i] = a[i]), Scale (b[i] = s x c[i]), Add (c[i] = a[i] + b[i]), Triad (a[i] = b[i] + s x c[i])로 구성된다. 실험에서 사용한 하나의 Intel KNL 노드는 68개의 프로세서 코어, 16GB 크기의 MCDRAM, 96GB 크기의 DDR4로 구성되어 있다. 메모리 모드는 Flat으로, 클러스터 모드는 Quadrant로 설정하였다. OpenMP를

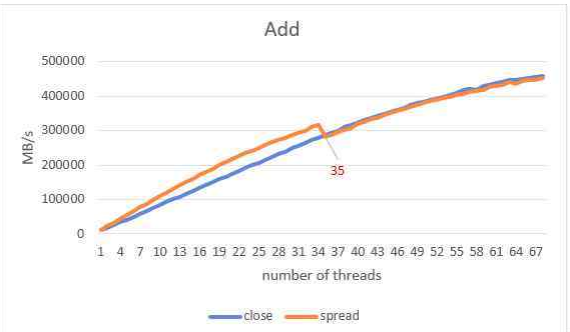
사용하여 스레드의 개수를 증가시켜가며 STREAM 벤치마크를 병렬 실행하였는데, 스레드가 실행될 위치를 지정하는 환경 변수 OMP_PLACES, 스레드가 배치되는 방식을 지정할 수 있는 OMP_PROC_BIND의 값들을 지정하여 실행하였다.

- 우선 스레드의 배치 방식이 성능에 미치는 영향을 알아보는 실험을 수행하였다. 환경변수 OMP_PLACES를 cores로 지정하면 스레드가 하나 이상의 하드웨어 스레드 들이 탑재된 프로세서 코어에 배치되고, socket으로 지정 하면 하나 이상의 코어로 구성되는 소켓에 배치된다. 실험에서는 cores로 지정하였다. OMP_PROC_BIND의 값을 close로 지정하면 i-번째 코어에 i-번째 스레드를 배치하여 스레드들이 가깝게 배치된다. Spread로 지정하면 스레드들을 분산시키는데, 전체 코어의 수를 스레드의 수로 나눈 값과 가까운 값을 간격으로 두어 배치한다. 실험에서는 close, spread를 모두 사용하였다. [그림2]는 STREAM의 각 연산에 대해 close와 spread를 비교한 실험 결과를 보여준다.



(a) Copy 성능

(b) Scale 성능



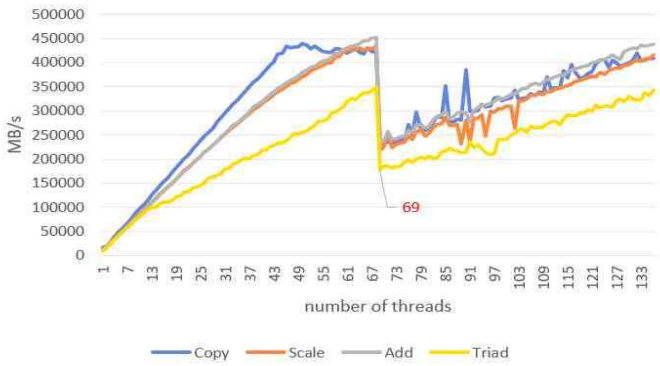
(c) Add 성능

(d) Triad 성능

[그림 2] 스레드 배치 방식에 따른 STREAM 벤치마크의 성능

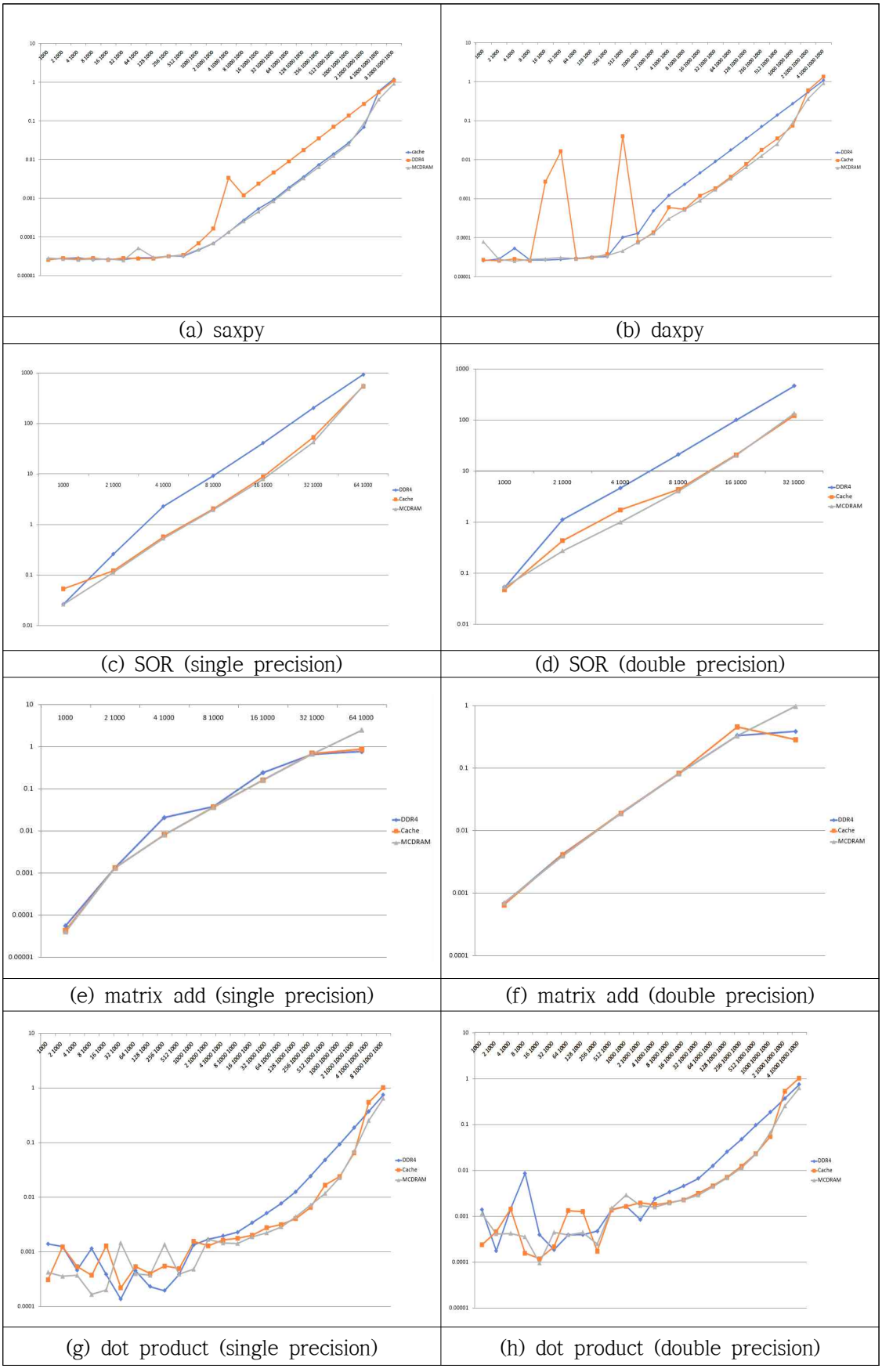
- 실험 결과 1~34까지는 spread 방식이 close 방식에 비해 우수한 성능을 보인다. Spread를 사용하면 34-스레드까지는 각 타일에 1개의 스레드만 배치되어 L2 캐시와 메모리 대역폭을 독점하여 사용하기 때문이다. 스레드 개수가 34를 넘어서 35가 되면 spread의 성능이 하락했다가 점차 다시 향상되는데, 궁극적으로는 close의 성능과 거의 같아진다. 스레드 수가 증가하면서 각 스레드당 할당되는 데이터의 양이 줄어들어 실행시간이 단축되는 반면, 35-스레드부 터는 close와 마찬가지로 각 타일에서 두 개의 코어들을 모두 사용하기 시작하면서 L2 캐시와 메모리 대역폭이 두 개의 코어들에 의해 공유되므로 하나의 코어가 독점하여 사용하던 경우에 비해 성능이 저하된다. 35-스레드의 성능 하락 현상은 후자의 성능 저하가 전자의 성능향상 효과를 더 크게 상쇄하기 때문이다.

- Spread 방식을 사용한 35-스레드에서의 성능 하락 현상은 연산별로 다르게 나타나는데, Copy 연산의 경우에 성능 하락이 가장 크게 나타나며, 하락한 성능이 다시 상승하여 close의 성능과 같아지는 지점도 늦게(스레드 개수가 ~55개가 되어서야) 나타난다. 반면 곱셈을 사용하는 Scale 연산의 경우 close 방식과 spread 방식에서의 성능이 거의 같으며 스레드 개수가 35일때 성능이 하락하는 현상이 나타나지 않는다. Add와 Triad 연산들에서도 35-스레드에서 성능 저하가 나타나는데, 이때의 저하된 성능이 close 배치 방식의 성능과 같아지며, 스레드 개수가 늘어도 같은 성능이 유지됨을 볼 수 있다.
- Spread를 사용할 경우 Copy 연산의 35-스레드에서의 성능 하락이 다른 연산들에 비해 크게 나타나는 이유는 메모리 접근 이외에 추가적인 연산이 없기 때문이다. 따라서 적은 수의 스레드로도 높은 성능을 보이지만 L2 캐시 와 메모리 대역폭이 두 개의 스레드들에 의해 공유되기 시작하는 35-스레드부터 성능 하락이 크게 나타나는 것이다. 반면 Scale의 경우 Copy 연산 이외에 곱셈 연산을 수행하기 때문에 스레드 개수가 늘어나면 곱셈의 성능이 향상되면서 Copy 연산에서 하락한 성능을 상쇄하게 된다. Add, Triad에서는 세 개의 배열들을 사용하여 Copy, 덧셈, 곱셈, 등의 연산을 수행하는데, Scale에 비해 더 많은 메모리 접근이 일어나고, 특히 Triad의 경우 Scale에 비해 추가로 덧셈 연산이 일어나므로 성능이 더 하락한다.
- 두 번째 실험에서는 스레드 개수를 1개부터 136개까지 증가시키며 STREAM 벤치마크를 수행했다. [그림 3]은 STREAM의 각 연산에 대한 실험 결과를 보여준다. 환경 변수 OMP_PLACES는 cores, OMP_PROC_BIND는 close를 각각 지정하였다. 실험 결과, 모든 연산에서 68-스레드까지는 성능이 향상된다. 그러나 69-스레드에서 크게 하락하고 136-스레드까지 다시 향상된다. 69부터 스레드의 수가 코어의 수보다 많아지면서 코어당 두 개의 스레드들이 할당되는 타일에서 처리하는 데이터의 양이 그렇지 않은 타일의 데이터 양을 크게 초과하여 전체 실행시간을 증가(전체 성능 저하)시키기 때문이다.



[그림 3] 스레드 개수에 따른 STREAM 연산 별 성능

- 또한 코어당 두 개 이상의 스레드들이 할당되면서 코어, L2 캐시, 메모리 대역폭, 등을 나누어 사용하기 때문이기도 하다. 그러나 스레드의 수가 늘어나 스레드당 할당되는 데이터의 양이 줄면 타일당 할당되는 데이터의 양이 전체적으로 비슷해지고 타일들의 부하가 균형이 맞추어지면서 다시 성능이 향상된다. 136-스레드의 성능은 68-스레드 성능보다 약간 낮은 수준까지 향상되는데, 코어 개수의 최대 2배에 달하는 스레드들이 각 타일의 메모리 대역폭 등을 공유함으로 인한 성능저하 정도가 크지 않음을 보여준다고 할 수 있다.



(a) saxpy (b) daxpy

(c) SOR (single precision) (d) SOR (double precision)

(e) matrix add (single precision) (f) matrix add (double precision)

(g) dot product (single precision) (h) dot product (double precision)

[그림 4] KNL 상에서 메모리 사용이 성능에 미치는 영향

2) KNL에서의 메모리 사용이 성능에 미치는 영향 연구

- 마이크로 벤치마크들(axpy, SOR, matrix add, dot product)을 활용한 성능 실험을 통하여, 아래와 같은 메모리 사용 방법들이 성능에 미치는 영향을 분석 :

  1) MCDRAM을 cache mode로 사용

  2) DDR4만 사용

  3) 메모리를 MCDRAM에 우선 할당(초과하면 DDR4에 할당)

- 실험을 위하여 CDRAM 사용의 효과를 알아본다.

  - 사용하는 코어의 개수/스레드 개수 : 64/64

  - clustering mode : Quadrant

  - 배열 크기 : 1000 ~ 16 x 1000^3 (x 데이터 타입에 따른 크기)

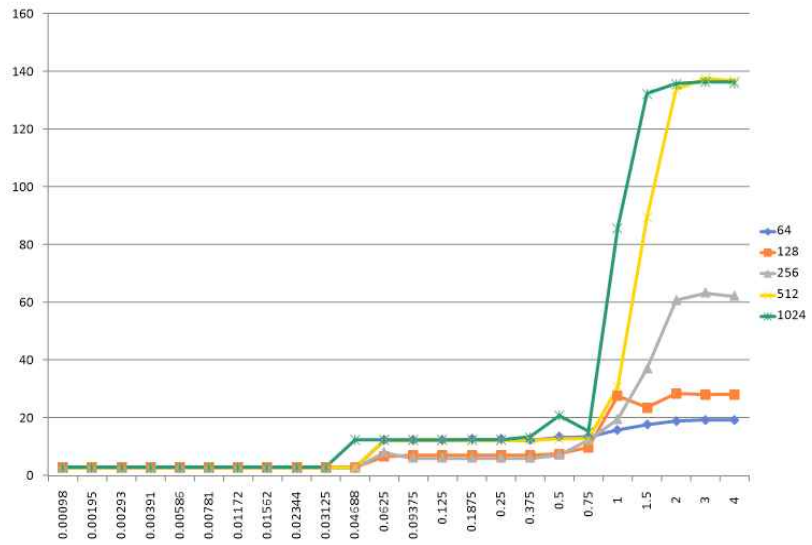
- 실험 결과 그래프([그림 4] 참조) 상에서 x-축은 배열의 크기를 나타내고(log 스케일로), y-축은 실행시간을 나타낸다(log 스케일).

- 전체적으로 MCDRAM을 사용하는 (1), (3)의 경우, 배열의 크기가 MCDRAM의 크기를 초과하기 전까지는 더 좋은 성능을 보이다가 그 이후부터는 DDR4와 비슷한 성능을 보이는 결과가 나타났다. 배열의 크기가 작을 때는 성능의 차이가 크지 않다.

- Cache mode에서는 MCDRAM을 last-level cache로 사용하는데, directed-mapped cache 방식으로 동작하기 때문에 특정 배열 크기에서는 매번 새로운 데이터를 필요로 하는 saxpy, 등과 같은 메모리 접근 패턴의 경우, cache miss를 처리하고 메모리에서 새 cache line을 가져오는 시간이 늘어나는 cache thrashing이 발생하여 성능이 저하된다.

- MCDRAM을 (3)의 방식으로 사용하는 경우에, 배열의 크기가 MCDRAM을 초과하면 성능이 z게 저하되는 현상이 나타난다(matrix add의 경우).

- 각 메모리의 latency를 측정하는 lm_bench를 활용한 성능 실험을 통하여, level-1 캐시, level-2 캐시, DRAM의 접근 시간을 측정. 실험 결과 그래프([그림 5] 참조) 상에서 x-축은 배열의 크기를 나타내고(GigaByte로), y-축은 실행시간을 나타낸다(nsec로).



[그림 5] lm_bench 성능 결과

3. 응용 프로그램의 성능 최적화

1) 가속기 상에서 Kokkos를 활용한 Cholesky Decomposition 알고리즘의 성능 최적화

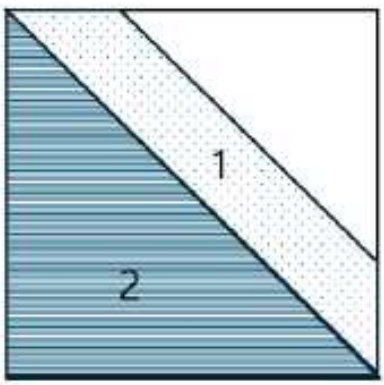
- Cholesky 분해는 양의 정부호 Hermite 행렬 A를 하삼각행렬(L)로 분해하는데, 양의 정부호 행렬 A를 하삼각행렬(L)과 하삼각행렬의 전치행렬로 분해하여 이들의 곱으로($A = L \cdot L^T$) 만들거나, 상삼각행렬과 상삼각행렬의 전치행렬로 분해하여 이들의 곱으로($A = U^T \cdot U$) 만들 수 있다. 이 경우 대각선 원소들의 값과 그 밖의 원소들의 값을 알해와 같이 구할 수 있다 :

$$u_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} u_{ki}^* u_{ki}} \qquad u_{ij} = \frac{1}{u_{ii}} \left(a_{ij} - \sum_{k=1}^{i-1} u_{kj}^* u_{kj} \right)$$

- Cholesky 분해 알고리즘에서는 아래의 [그림 6] 코드에서 보듯 삼중 반복문(for loop)을 통해 원소 하나씩을 하삼각행렬의 크기만큼 L[0,0]부터 시작하여 왼쪽 열부터 대각에 위치한 원소까지 계산하는 구조인데([그림 7] 참조), 이 과정에서 다음과 같은 집중적인 행렬 곱셈 연산이 요구됨. 이처럼 다양한 원소에 지속적으로 메모리에 접근해야 하는 알고리즘 특성상 병렬처리를 통한 성능향상이 절실하다.

```
for i from 0 to n do
    for j from 0 to i do
        s = 0
        for k from 1 to j-1 do
            s = s + L[i,k] * L[j,k]
        end for
        if i == j then L[i,j] = sqrt(A[i,i] - s)
            else L[i,j] = (1.0 / L[j,j]) * (A[i,j] - s)
        end if
    end for
end for
```

[그림 6] Cholesky Decomposition Pesudo 코드

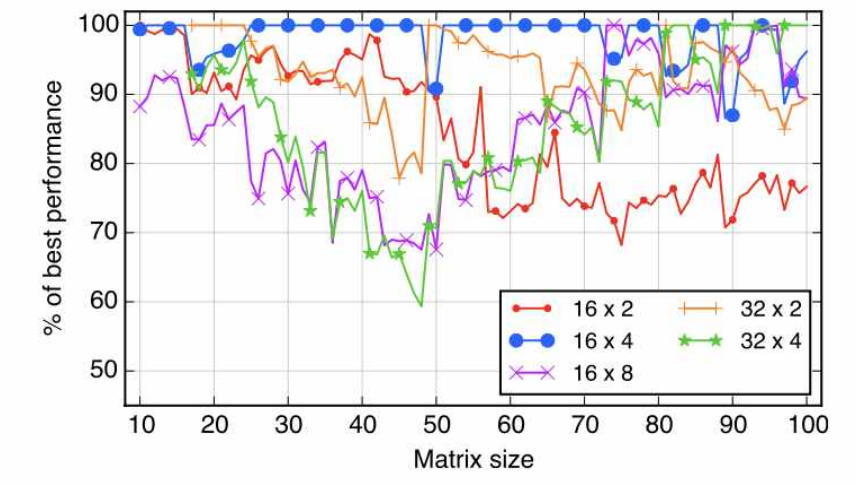


[그림 7] 병렬 Cholesky 분해 과정

- 본 연구는 GPU 상에서 kokkos를 활용하여 Cholesky 분해를 병렬처리함으로써 성능을 최적화하는데, kokkos의 특성과 GPU의 구조적 특징을 잘 활용하면 이러한 목표를 달성할 수 있음.
- Kokkos에서는 Kokkos::parallel_for, Kokkos::parallel_reduce 등의 API를 통해 GPU 병렬화를 간결한 코드로 최적화 가능하며, Kokkos::View, Kokkos::deep_copy 등을 활용해 메모리 관리를 추상화하고, 자동 메모리 배치 최적화 (Kokkos::LayoutRight vs Kokkos::LayoutLeft)를 지원하며, Kokkos::Cuda, Kokkos::OpenMP를 활용하여 실행 공간을 변경함으로써 CPU/GPU 간 코드가 공유 가능하도록 한다. Cholesky 분해 병렬화를 위하여 Kokkos::View를 활용하였는데, 다차원 배열에 대해 편리한 메모리 할당과 관리가 가능해진다. , 병렬처리를 위한 데이터 통신 최적화를 기대할 수 있다.
- GPU 아키텍처의 특성을 활용하기 위하여 소프트웨어 블록의 개수, 블록 당 쓰레드 개수 조절을 통하여 Cholesky 분해 연산 프로그램을 GPU 가속기에 매핑하는데, blockId와 threadId 를 이용하여 우선적으로 대각 위치의 코어를 먼저 제곱근을 취하고 이전 원소들을 이용해 계산함. 이후 대각 요소 기준으로 삼각행렬의 각 요소를 나누는 작업을 수행한다. 대각에 위치한 원소 하단의 삼각행렬 원소들은 전체 행렬의 크기를 스레드의 수로 등분하여 처리할 작업의 범위를 기준으로 각자 담당한 원소 연산을 완료하여 병렬화를 극대화함.
- Kokkos 기반 병렬(CUDA backend 활용) 코드를 구현하고 실행시간을 측정하여 직렬 Cholesky 분해 코드의 실행시간과 비교하였다([표 1] 참조). 실험을 위해 1,000x1,000 크기의 양의 정 부호 Hermite 행렬을 생성해 각 프로그램별 커널 실행시간을 비교하여 성능 향상 정도를 분석하였다. 실험 결과 Kokkos View를 활용하여 순차 실행 프로그램 대비 약 500배에 달하는 성능 향상을 얻었다.

[표 1] Cholesky 분해의 순차, 병렬 실행 시간 비교

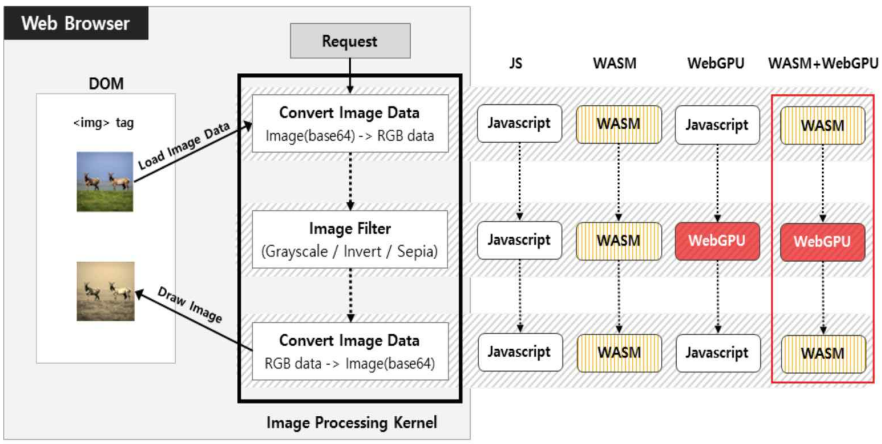
	Serial	Kokkos-CUDA
Time(ms)	15.459	0.031
Speedup(Ts/Tp)	1	498.68



[그림 8] (블록의 개수 x 블록당 스레드 개수)와 성능

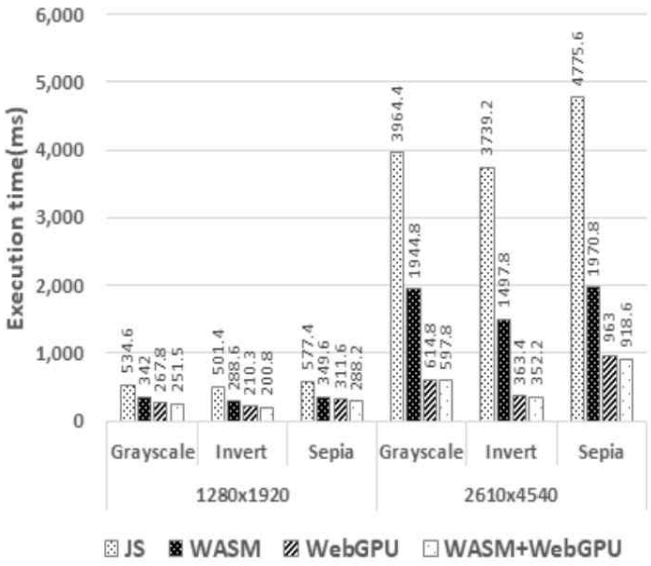
- [그림 8]는 다양한 블록 수-쓰레드 수 크기를 설정하여 Nvidia RTX 4070ti GPU 상에서의 가속화 정도를 측정한 결과를 나타내며, 행렬의 크기에 따라 적절한 블록의 개수와 블록 별 쓰레드 개수가 존재함을 알 수 있음. 1000x1000 크기의 행렬에서는 블록 개수가 16개, 블록당 스레드의 개수가 4일 때 가장 성능이 우수함을 관찰 할 수 있다([그림 34] 참조).
- 연구 결과는 한국정보처리학회 추계학술대회 ACK2024(2024-10)에 발표됨(논문 제목 : KOKKOS 환경에서의 병렬 숄레스키 분해 구현).

2) 웹 기반 이미지 프로세싱 응용의 성능 최적화 연구



[그림 9] WebGPU 및 WASM을 활용한 이미지 프로세싱 응용

- 고성능이 요구되는 과학기술, 인공지능, 이미지/비디오 프로세싱 응용 프로그램들이 웹 기반 애플리케이션으로 개발되고 있다. 과거 이들은 비표준 웹 플러그인 기술이나 서버 기반 프로세싱 방식을 사용해왔으나, 최신 웹 브라우저에서의 지원 중단, 서버 사용량 증가에 따른 비용 증가, 등의 문제점들이 있다. 이러한 문제점들을 해결하고자 최신 웹 브라우저에서 제공하는 GPU를 직접 활용하는 WebGPU 표준과 CPU에서 고속 실행을 보장하는 WASM (Web Assembly) 표준을 함께 적용하여 웹 브라우저 기반 알고리즘의 성능을 최적화한다.
- 먼저, 이미지 프로세싱을 위한 응용의 성능을 향상시키는데, CPU, GPU를 모두 활용할 수 있는 형태로 만들어 최적의 성능을 얻을 수 있도록 한다. 이미지를 RGB로 변환하거나 복원하는 부분들은 JavaScript로 보통 많이 작성되는데, 이 부분들을 C, C++, Rust 등의 언어들을 사용하여 작성하고 Emscripten 컴파일러를 사용하여 WASM 코드로 변환한다. 계산량이 많은 필터링 부분은 WASM으로 작성하거나 WebGPU 코드로 작성한다. WebGPU로 작성하기 위하여 WGSL (WebGPU Shader Language) 표준 셰이더 언어인 Rust를 사용한다. 웹 페이지로부터 이미지를 로드하거나 전/후처리하는 부분들은 Host 영역을 접근해야 하므로 WASM 코드로 작성한다. 이렇게 작성된 WebGPU 및 WASM 코드들을 일반 웹 페이지 코드인 HTML, CSS, Javascript 코드와 병합하여 하나의 Web App으로 패키징한다([그림 9] 참조).

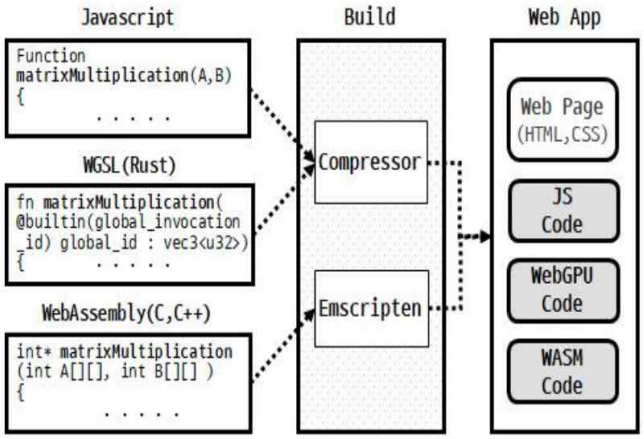


[그림 10] 이미지 프로세싱 응용의 실행시간 비교

- 이미지 프로세싱 응용의 전처리 단계, 필터링 처리 단계(Grayscale, Invert, Sepia 등의 알고리즘 활용), 후처리 단계에서 각 단계의 세부 로직들을 JaveScript만 이용하여 구현, WASM만 이용하여 구현, JavaScript+WebGPU을 이용하여 구현, WASM+WebGPU을 이용하여 구현한 경우들의 성능을 비교 평가한다([그림 10] 참조). 실험에서는 Intel i5-10210U (1.6Ghz) CPU와 Intel의 UHD Graphics (8Gb) GPU를, 웹브라우저는 Chrome 116.0.5845.188을 사용한다. 실험 결과 WebGPU의 성능(JavaScript+WebGPU 또는 WASM+WebGPU)이 JavaScript에 비해서는 물론, WASM의 경우와 비교해도 크게 향상되는 것을 볼 수 있다. 특히 WASM+WebGPU의 경우 성능이 가장 크게 향상되는 것을 볼 수 있다. 성능 향상의 정도는 고성능이 요구되는 정밀도가 높은 이미지의 경우 더 큰 것으로 나타난다.
- 연구 결과는 한국정보처리학회 논문지(2024-10)에 발표됨(논문 제목 : 웹GPU와 웹어셈블리를 이용한 이미지 프로세싱 가속).

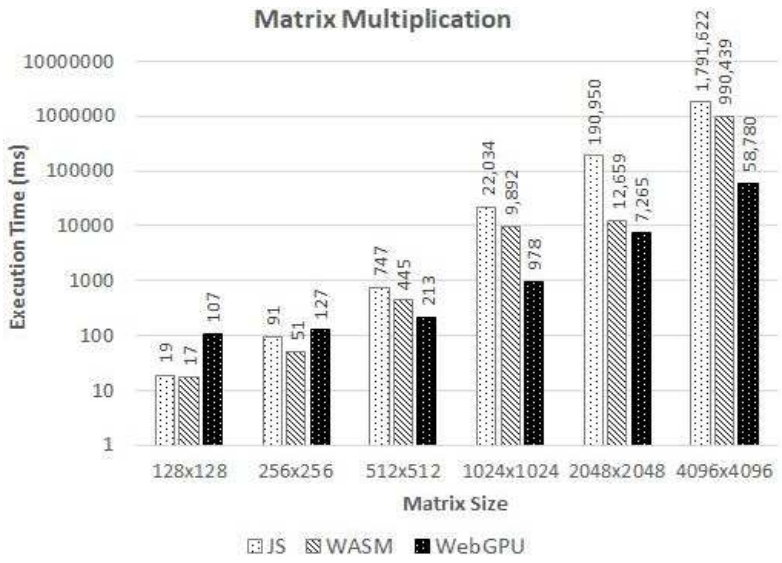
3) 웹 기반 행렬 곱셈의 성능 최적화 연구

- 다음으로 행렬 곱셈의 성능을 향상시키는데, 최근 웹브라우저는 인공지능 등의 응용 분야에서도 널리 활용되고 있다. 예를 들어 웹브라우저 기반 인공지능 애플리케이션을 구현하는 경우, 인공 신경망에서 처리되는 계산들을 행렬 곱셈을 활용하여 처리한다. 행렬 곱셈의 성능 향상을 위해 이미지 프로세싱에서와 같은 접근법을 활용한다(CPU 상에서 실행 가능한 Javascript 및 WASM 코드, GPU에서 실행 가능한 WGSL 코드 작성). 먼저 각 코드 타입별로 소스코드를 작성하는데, WebGPU는 Rust 기반의 WGSL 셰이더 언어로 작성하고, WebAssembly는 C, C++ 언어로 작성하고 Emscripten 컴파일러를 이용하여 WASM 코드로 빌드한 후 패키징된다([그림 11] 참조).
- 다양한 크기의 행렬들을 사용하여 구현한 경우들의 성능을 비교 평가한다([그림 12] 참조). 실험에서는 Intel i5-10210U (1.6Ghz) CPU와 Intel의 UHD Graphics (8Gb) GPU를, 웹브라우저는 Chrome 116.0.5845.188을 사용한다. 행렬의 크기가 작은 128x128, 256x256에서는



[그림 11] 행렬곱셈의 Web 실행 코드 작성 과정

WASM 코드의 실행 속도가 가장 빠르고 WebGPU 코드의 경우는 오히려 JS 보다도 느린데, 이는 초기화 시점에 WebGPU 객체를 생성하고 커널 코드에 인자값 전달 및 계산 결과를 받아올 때 CPU와 GPU 사이의 데이터를 복사하는 과정에서 발생하는오버헤드가 전체 실행시간에서 차지하는 비중이 크기 때문이다. 행렬의 크기가 1024x1024, 2048x2048, 4096x4096와 같이 커지는 경우 WebGPU 코드의 실행 속도가 JS 코드에 비해 최대 30배, WASM에 비해 ~17배 빨라진다. 작업량이 커질수록 연산 실행시 초기화, 데이터 송/수신 작업에 소요되는 시간이 전체 실행시간에서 차지하는 비중이 줄어들기 때문이다.



[그림 12] 행렬 곱셈 실험 결과

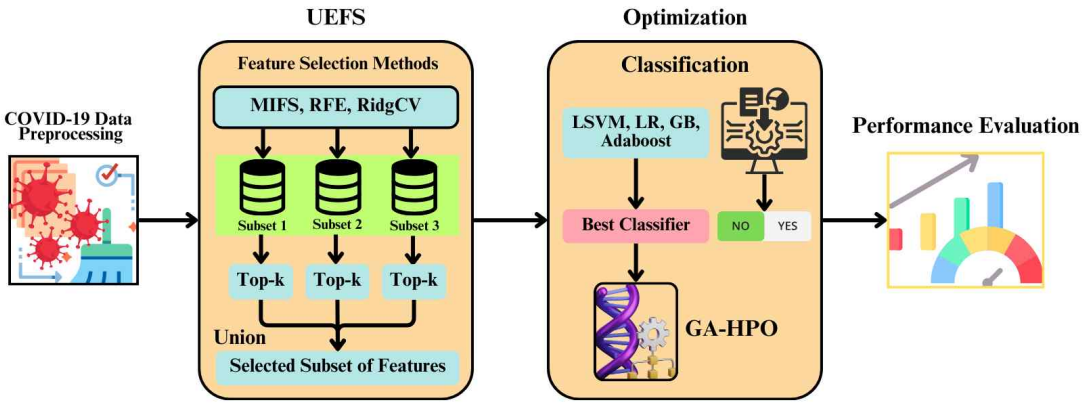
- 연구 결과는 한국정보처리학회 추계학술대회 ACK2024(2024-10)에 발표됨(논문 제목 : 웹 실행 코드 타입에 따른 행렬 곱셈 성능 평가).

4) 인공지능 응용의 성능 최적화

- 기계학습(Machine Learning: ML) 기법을 이용하여 환자들을 진단할 수 있는 의사결정 기술들이 최근 개발되었는데, ML 기법을 이용하면 전통적인 의료기술의 제약(검사 및 증상을 통한 진단에 오랜 시간

이 걸리는)에서 벗어나 효율적으로 빠른 진단을 내릴 수 있는 것이 장점임. 이러한 ML 기법은 의료 리포트를 학습하여 의사 결정을 내리고, 질병을 자동으로 예측하는 분석 도구임.

- 연구에서는 ML 기법을 이용하여 COVID-19 감염 환자들을 식별하는 자동 진단 방법 개발. 2019년말 등장한 COVID-19 호흡기 질환은 전 세계적으로 빠르게 확산되어 약 3년간 전 세계적으로 약 7억 6,100만 명에게 피해를 줌. 혈액 검사, 영상 기법, 중합효소연쇄반응(PCR) 검사 등 다양한 코로나19 검출 방법이 사용됐지만, 사망률 최소화를 위해 절실한, 감염된 개인을 우선적으로 식별하는 데에는 한계를 보임. 기계학습 기법을 이용하여 감염 환자를 초기 단계에서 식별하는 간결하고 정확한 자동 진단 방법을 개발.
- COVID-19을 자동으로 식별하기 위한 이전의 연구들은 높은 예측 정확도로 질병을 분류하지만 대부분은 기능이 적고 크기가 작은 데이터 세트를 사용함으로써 과적합, non-Gaussian 노이즈 발생, 등의 문제점들을 안고 있음. 가치 있는 feature들을 추출하기 위해 개별적인 feature selection (FS) 기법이 사용되기도 했으나, 데이터와 모델의 동시 최적화를 고려한 유용한 프레임워크를 만드는 것이 필요.
- 최근 앙상블 학습 기법에 영향을 받은 앙상블 FS (EFS) 기법이 개발되었는데, EFS 접근 방식은 개별 FS 기술보다 성능이 뛰어나며, 최적의 기능 조합을 생성할 수 있는 다양한 기능 하위 집합을 생성. 연구에서는 COVID-19 예측 정확도를 높이기 위해 EFS를 개선한 최적화된 통합 앙상블 FS (OUEFS: Optimized Union Ensemble Feature Selection) 접근 방식을 개발함([그림 13] 참조). OUEFS는 FS 프로세스를 통해 얻은 기능 하위 집합의 통합



[그림 13] OUEFS (Optimized Union Ensemble Feature Selection) 접근법

앙상블을 사용하며, ML 분류기의 성능 최적화를 포함함.

- OUEFS에서는 먼저 데이터 전처리 단계에서는 누락된 값, 데이터 불균형 및 정규화 해결을 포함한 임상 증상 데이터 세트에 대한 정확한 데이터 전처리를 수행하는데, 이를 통해 MIFS, RFE, RidgeCV 등의 FS 전략을 적용하여 핵심 특징을 식별. 가장 중요한 기능 하위 집합은 Top-k 임계값 기술을 사용하여 선택. 이렇게 선택된 기능 하위 집합은 다양한 통합 조합을 통해 결합되어 최적의 앙상블 기능 하위 집합을 만듬. 그런 다음 LSVM, LR, GB 및 Adaboost의 4가지 ML 분류기를 사용하여 분류를 수행. 분류를 통해 환자의 감염 여부를 예측하고 앙상블 기능 하위 집합을 기반으로 가장 성능이 좋은 분류기를 식별. 선

택한 분류기의 성능을 더욱 최적화하기 위해 Genetic Algorithm을 사용하여 하이퍼파라미터를 미세 조정. 마지막으로 정확도, 정밀도, 재현율, F1 점수 및 AUC 지표를 사용하여 모델 성능을 평가.

Algorithm 1 : Interpretable KD Prediction Model

Input:
Preprocessed KD dataset D, Number of selected features k,
Classifier set $\{C_i\}_{i=1}^{n}$

Output:
Best-performing classifier model C', Interpretable explanations

Step 1: Data Preprocessing

Normalize dataset D using min–max scaling:

$$D' = \frac{D - D_{min}}{D_{max} - D_{min}}$$

Balance classes in D' to address class imbalance using SMOTE.

Step 2: Feature Selection using ACO

Initialize ant colony optimization parameters: population P, iterations T, evaporation rate ρ.

Calculate the importance score $S(f_i)$ for each feature $f_i \in F$.

Update pheromone τ_{ij} for feature subsets F_t based on scores:

$$\tau_{ij} = (1 - \rho).\tau_{ij} + \rho.S(f_i)$$

Select subset F_k containing the top k features with the highest $S(f_i)$:

$$F_k = top - k(F, S)$$

Step 3: Classifier Training and Evaluation

For each classifier $C_i \in (C_1, C_2, \ldots, C_n)$

C_i on F_k using an 80:20 training–validation split.

k-fold cross-validation to optimize performance metrics M (e.g., accuracy, precision, recall, F1-score, Kappa, MCC):

$$M_i = evaluate(C_i, F_k)$$

Select $C' = argmax_{C'} M_i$

Step 4: Explainability Analysis using SHAP and LIME

Initialize SHAP and LIME explainers.

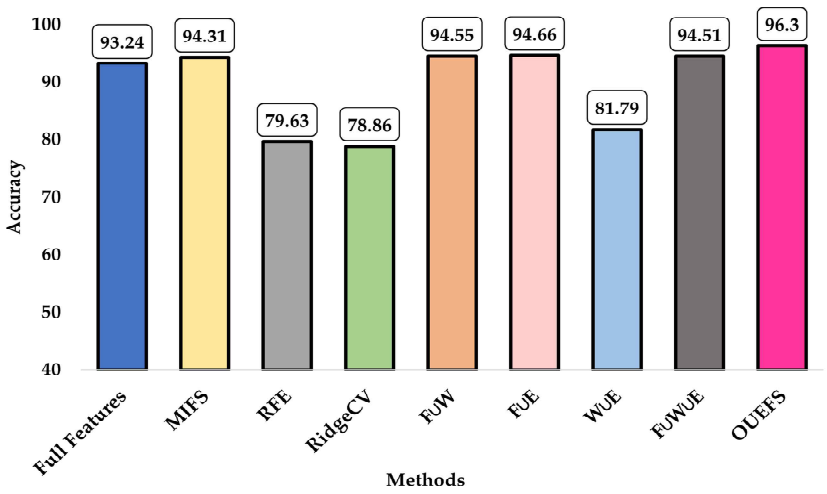
For each instance x in D' compute, SHAP values *SHAP(x)* and LIME explain *LIME(x)* to analyze the impact of features on predictions.

Return the best-performing model C' and local/global interpretability results from SHAP and LIME.

Return: C' with interpretability analysis.

[그림 14] 제안하는 OUEFS 프레임워크의 pseudocode

- 실험 결과는 Adaboost 분류기 및 GA-HPO와 함께 MIFS 및 RidgeCV FS 기술의 통합 앙상블이 96.30%의 정확도를 달성한 것으로 나타남. 96.30%의 정확도는 COVID-19 예측을 위한 이전의 모든 앙상블 기반 방법보다 성능이 뛰어남([그림 15] 참조).
- 연구 결과는 SCI 학술지인 IEEE Access, 2024년 07월호에 게재됨(제목 : High Accuracy COVID-19 Prediction Using Optimized Union Ensemble Feature Selection Approach).



[그림 15] 이전 연구들과 제안하는 OUEFS의 예측 정확도 비교