[2024-3]

KEASE 기술서

한국형 엑사스케일 응용 SW 개발 환경 프레임워크

슈퍼컴퓨터 5호기의 성능모델 및 프로그램 실행시간 예측모델 개발

2024. 04. 20

명 지 대 학 교

[개 정 이 력]

개정 일자	버전	설명	개정페이지	작성자	확인자
2024.4.20	1.0	INITIAL DRAFT	전체	이명호	이명호

목 차

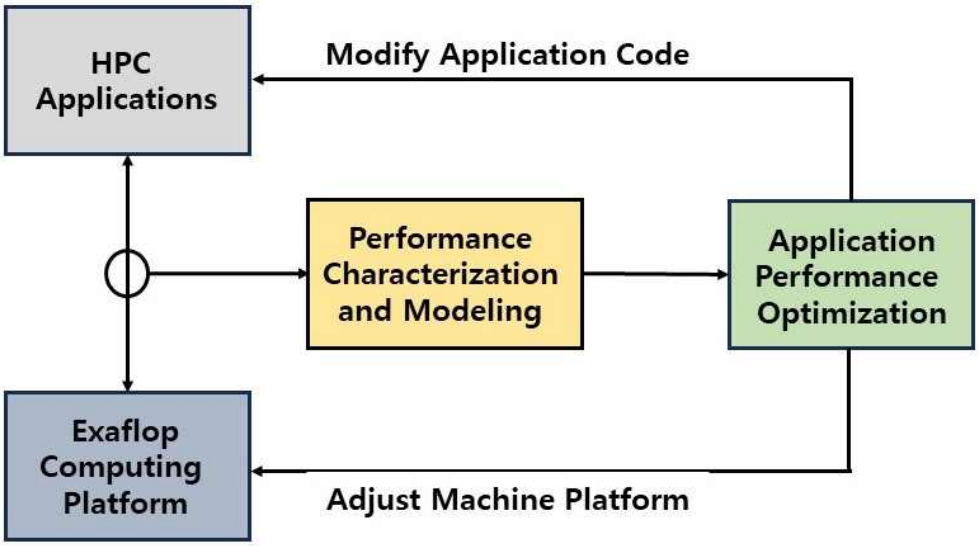
1. 연구의 개요

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 특징을 모델링하고, 이러한 모델에 기반하여 병렬화, 최적화 기술을 개발한다. 먼저 Intel KNL 아키텍쳐의 성능적 특성을 고수준에서 모델링하는데, 특히 가속칩의 주요 아키텍쳐 parameter들을 이용한 수식을 도출함. 그런 다음, 응용 프로그램의 병렬 실행 가능 영역의 하드웨어 자원 요구량 등을 추출하여 프로그램을 모델링하고, 슈퍼컴 아키텍쳐 상에서의 응용 프로그램 실행 시간 예측 모델을 함께 개발한다.

아키텍쳐 및 응용 프로그램 모델에 기반한 병렬화 및 성능 최적화 기술들을 응용 프로그램에 적용하여 성능을 최적화한다. 전통적인 HPC 응용 및 최근 활용도가 높아진 인공지능 응용을 모두 타켓으로 하여 병렬화 및 성능 최적화 연구를 수행한다. [그림 1]은 본 연구의 개요를 나타낸다.

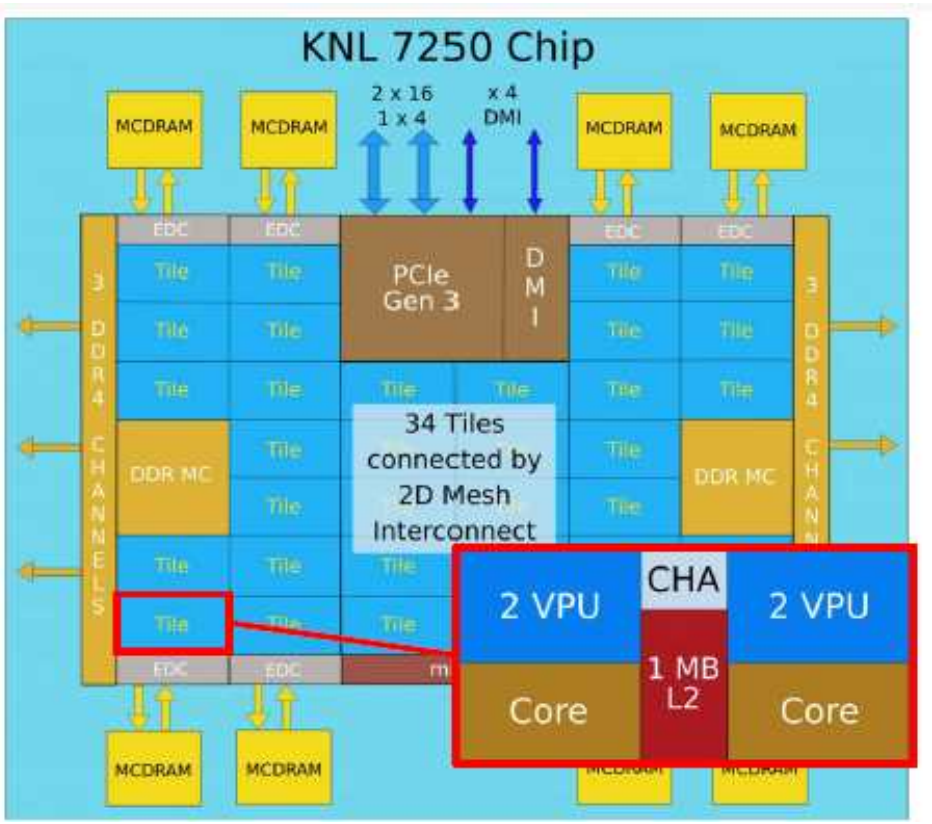


[그림 1] 아키텍쳐 및 응용 프로그램 모델링에 기반한 응용의 성능 최적화 개요도

2. 슈퍼컴퓨터 5호기의 성능모델 및 프로그램 실행시간 예측모델 개발

1) 슈퍼컴 5호기의 Intel KNL 아키텍쳐 성능 모델 및 응용 프로그램의 실행시간 예측 모델

- Intel KNL (Knight Landing) 아키텍쳐의 개요
  - Intel KNL은 64~72개의 silvermont 코어들로 이루어진 MIC (Many Integrated Core) 칩이다. 각 코어는 1.3~1.5GHz로 작동한다. 2개의 코어들은 하나의 tile을 구성한다 ([그림 2] 참조).
  - KNL은 이전 세대인 Kights Corner (KNC)에서의 coprocessor 모드가 아닌 stand-alone 방식으로 수행이 가능하도록 설계되어 보다 유연한 사용이 가능하다.
  - 펌웨어 세팅을 통해 코어들의 클러스터링 구조(코어들을 어떤 방식으로 그룹화하고 연결할 것인지)를 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.
  - 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 2] Intel KNL 아키텍처

- Intel KNL 아키텍쳐의 성능 모델
  - 위와 같은 KNL 아키텍쳐의 특징을 모델링하면 크게 병렬성과 메모리 구조로 나누어 볼 수 있다.
  - 병렬성은 KNL 칩 내부에 존재하는 코어의 개수(N), 각 코어당 스레드의 개수(M)로 표현이 가능하다. 즉, N 개의 코어들이 병렬 실행을 수행하고, 각 코어 당 M개 (KNL의 경우에는 4개)의 스레드들이 멀티 스레딩(Hyper-Threading) 방식으로 동시에 스레드들을 실행한다. 따라서 동시 실행 가능한 스레드 수준의 병렬성을 N x M으로 표현할 수 있다.
  - 이에 더하여 각 코어 당 2개의 512-bit VPU들이 SIMD 방식의 병렬 실행을 수행한다. 만약 single-precision 연산을 실행할 경우 Ss (512/32=16)개의 SIMD 연산을, double-precision 연산을 실행할 경우 Sd (512/64=8)개의 SIMD 연산을 실행한다.
  - 위의 계산들을 종합하면, 스레드 수준의 병렬성과 SIMD 병렬성 Ss 또는 Sd를 곱하면 N x M x Ss 또는 N x M x Sd의 병렬성을 구할 수 있다.

- KNL 상에서의 응용 프로그램 실행시간 예측 모델
  - 위의 KNL 아키텍쳐의 성능모델은 스레드 수준의 병렬성과 SIMD 병렬성을 곱하여 N x M x Ss 또는 N x M x Sd의 병렬성을 구했다.
  - 만약 병렬화 가능한 for-loop의 iteration count가 IC라고 가정하면, loop의 실행 시간을 IC/(N x M x Ss) 또는 IC/(N x M x Sd)로 표현할 수 있다. 이때 모든 메모리 접근이 register와 L1 캐시에서 참조가 가능하다고 가정한다.
  - 메모리와 관련하여서는 레지스터, L1 캐시와 같이 아주 빠르고 접근 속도가 일정한 메모리들의 접근 시간을 t1으로 모델링한다. MCDRAM 또는 DDR DRAM의 접근 시간을 t2라고 가정한다.
  - 자주 사용되는 데이터들은 L1 캐시, L2 캐시 등에 캐싱되는데, 이들에 대한 접근 속도는 캐싱된 상태인지 아닌지에 따라서 크게 차이가 발생하며 t1~t2 사이에 들어간다고 볼 수 있다.
  - 이러한 메모리 모델을 고려한 프로그램의 실행 시간은 위의 실행시간(IC/(N x M x Ss/d))에 (t2 - t1) x (1 - h)를 곱한 식을 더한 것으로 표현할 수 있다. 여기서 h는 L1 캐시 메모리의 hit ratio를 말한다.

2) 슈퍼컴 5호기의 Intel KNL의 성능 실험

- KNL의 성능적 특성을 분석하기 위해 마이크로 벤치마크 프로그램들을 활용하여 아래와 같은 성능 실험들을 수행하였다:

Axpy 연산

- Axpy 연산은 길이 N의 벡터 X에 scalar 값(a)을 곱하여 또 다른 벡터 Y에 누적시키는 연산인데(Y[0:N-1] += a * X[0:N-1]), 벡터의 기본 데이터 타입이 single-precision float일 경우에는 saxpy, double-precision일 경우에는 daxpy로 불린다.

```
void saxpy(int n, float a, float *x, float *y) {
        #pragma omp parallel for shared(x, y, N) private(i, a)
         for (int i=0; I < N; i++)
                y[i] += a * x[i];
}
```

- N의 사이즈가 충분히 클 때(2 x 1,024 x 1,024 x 1,024), 64 코어를 모두 구동하여 saxpy의 경우 0.48sec, daxpy의 경우 0.58sec의 실행시간을 구하였다. 이때 scalability는 각각 59.2, 51.4에 이른다.
- 멀티 스레딩 기능을 활용하여 각 코어당 스레드의 개수를 2, 4으로 높이면 실행시간은 오히려 0.61sec, 0.55sec (daxpy는 0.66sec, 0.61sec)로 늘어난다. 따라서 코어당 하나의 스레드만 구동하는 것이 saxpy/daxpy 연산을 위하여 유리함을 알 수 있다.
- N이 이보다 작을 경우, 실행 시간이 너무 짧고, scalability가 일찍 꺾이는 점을 관찰할 수 있었다.

Dot product 연산

- 길이 N의 행 벡터 X와 같은 길이의 열 벡터 Y의 원소들을 곱하여 누적값을 구한다.

```
float dot_product(int N, ffloat *x, float *y) {
    #pragma omp parallel for shared(x, y, N) private(i) reduction(* : p);
    for (int i=0; I < N; i++)
        p += x[i] * y[i];
}
```

- N의 사이즈가 충분히 클 때(4 x 1,024 x 1,024 x 1,024), 64 코어를 모두 구동하여 single-precision dot product는 1.39sec, double-precision은 1.54sec의 실행시간을 구하였다. 이때 scalability는 각각 63.4, 58에 이른다.
- 멀티 스레딩 기능을 활용하여 각 코어당 스레드의 개수를 2, 4으로 높이면 실행시간은 오히려 1.71sec, 1.54sec (daxpy는 1.77sec, 1.55sec)로 늘어난다. 따라서 코어당 하나의 스레드만 구동하는 것이 dot product 연산을 위하여 유리함을 알 수 있다.
- N이 이보다 작을 경우, 실행 시간이 너무 짧고, scalability가 일찍 꺾이는데, 이는 곱셈 및 덧셈 계산 시간에 비하여 reduction 연산의 오버헤드가 상대적으로 크기 때문이다.

행렬 덧셈 연산

- N x N 크기의 행렬 A와 B의 각 원소들을 더하여 같은 크기의 결과 행렬 C를 구한다.

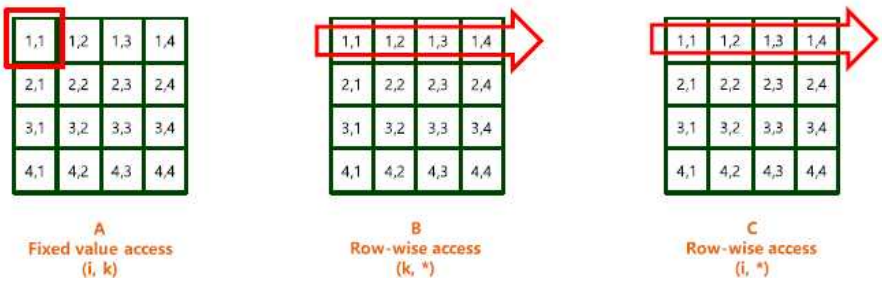
```
#define N      1024
main() {
float A[N][N], B[N][N], C[N][N];
int i, j;
    #pragma omp parallel for shared(A, n) private(i, j)
      for (i=0; I < N; i++)
          for (j=0; j< N; j++)
            C[i][j] = A[i][j] + B[i][j];
    }
```

- SIZE = 32,768 (single-precision float 일 때), 32 코어를 모두 구동하여 1.65sec의 실행시간을 구하였다. 이때 scalability는 각각 39에 이른다.
- A, B, C 행렬 모두 연속되는 데이터들을 접근하는 프로그램의 특성 및 data locality가 잘 반영되어 뛰어난 성능을 보임을 알 수 있다.

밀집 행렬 곱셈

- 두개의 SIZE x SIZE 크기의 행렬 A, B를 곱하여 결과 행렬 C를 구하는데, 삼중으로 중첩되는 i, j, k loop들을 아래와 같이 i-k-j 순서로 배치하여 캐시 메모리 및 각 코어의 VPU에 친화적이 되도록 한다.

```
for(int i=0; i<SIZE; i++) {
    for(int k=0; k<SIZE; k++) {
        double A_val = A[i][k];
        for(int j=0; j<SIZE; j++) {
            C[i][j] += A_val * B[k][j];
        }
    }
}
```



- SIZE = 4,096일 때, 64 코어를 모두 구동하여 2.33sec의 실행시간을 구하였다. Loop 순서를 i-j-k로 두면 같은 크기의 행렬 곱셈 실행시간이 10배 이상 증가하는데, i-k-j 순서가 캐시에서의 data locality를 잘 활용함을 알 수 있다.
- 같은 크기의 행렬 및 loop 순서를 이용하고, VPU 활용을 극대화하기 위하여 -simd 컴파일러 flag를 이용하여 실행파일을 생성한 결과 추가적으로 ~15%의 성능 향상을 얻었다.

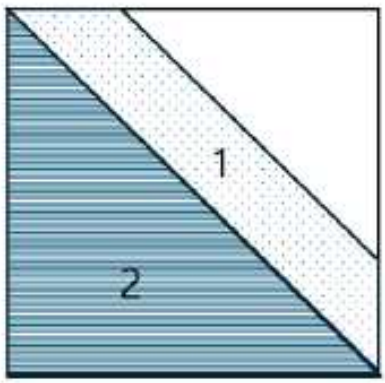
3. 응용 프로그램의 성능 최적화

1) 가속기 상에서의 Cholesky Decomposition 응용 프로그램 성능 최적화

- Cholesky 분해는 양의 정부호 행렬 A를 하삼각행렬(L)로 분해하여 $A = L \cdot L^T$ 로 만드는 연산임
- 이 과정에서 아래의 코드에서 보듯 삼중 반복문(for loop)을 통해 원소 하나씩을 하삼각행렬의 크기만큼 L[0,0]부터 시작하여 왼쪽 열부터 대각에 위치한 원소까지 계산하는 구조인데, 이 과정에서 집중적인 행렬 곱셈 연산이 요구된다.

```
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
```

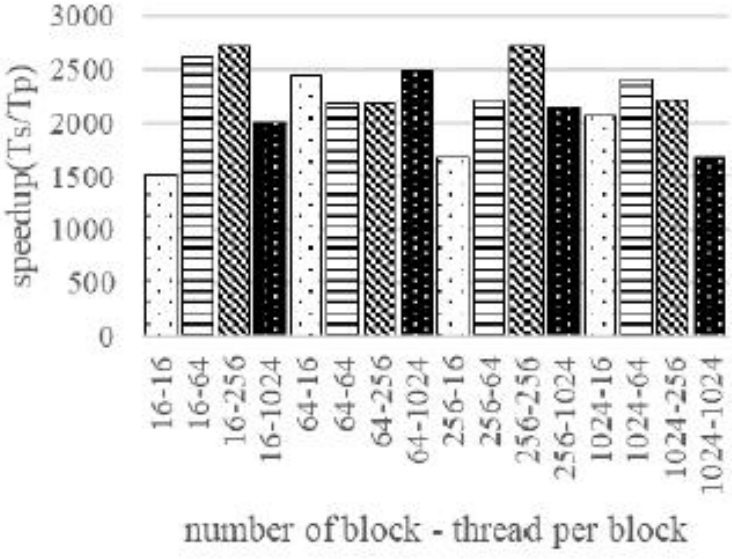
[그림 3] Cholesky Decomposition Pseudo 코드



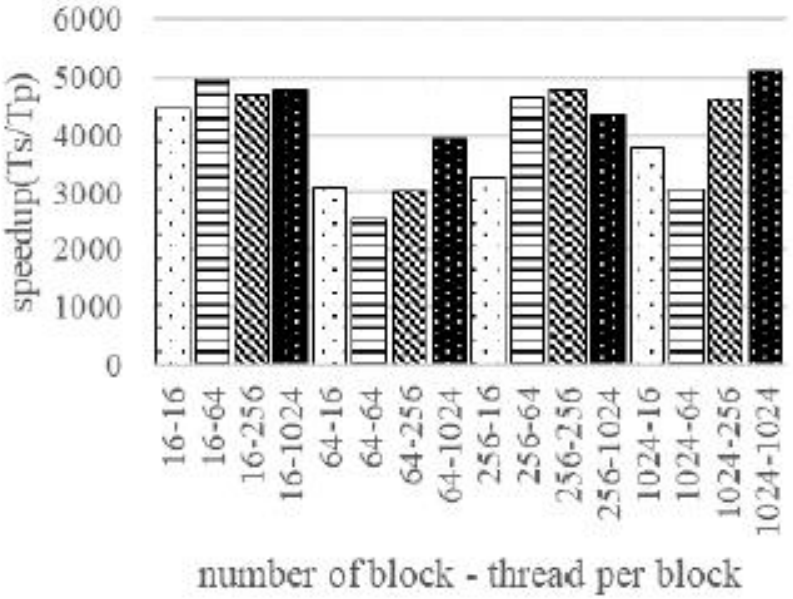
[그림 4] 병렬 Cholesky 분해 과정

- 본 연구에서는 가속기 상에서 Cholesky 분해를 병렬처리함으로써 성능 최적화를 꾀한다.
- GPU 가속기의 구조적 특징을 잘 활용하면 이러한 목표를 달성할 수 있음.
- 소프트웨어 블록의 개수, 블록 당 쓰레드 개수 조절을 통하여 Cholesky 분해 연산 프로그램을 GPU 가속기에 매핑하는데,
- blockId 와 threadId 를 이용하여 우선적으로 대각 위치의 코어를 먼저 제곱근을 취하고 이전 원소들을 이용해 계산한다.

- 그런 다음, 대각 요소 기준으로 삼각행렬의 각 요소를 나누는 작업을 수행한다.



[그림 5] 1,000 x 1,000 행렬의 속도 향상



[그림 6] 2,000 x 2,000 행렬의 속도 향상

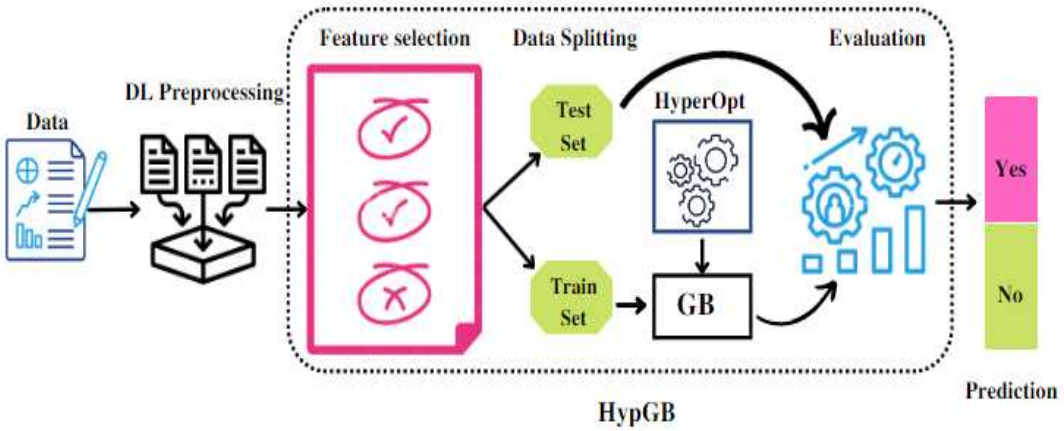
- 대각에 위치한 원소 하단의 삼각행렬 원소들은 전체 행렬의 크기를 스레드의 수로 등분하여 처리할 작업의 범위를 기준으로 각자 담당한 원소 연산을 완료하여 병렬화를 극대화한다.
- 다양한 블록 수-쓰레드 수 크기를 설정하여 Nvidia RTX 4070ti GPU 상에서의 가속화 정도를 측정한 결과를 [그림 5]와 [그림 6]에서 보여주는데, 행렬의 크기에 따라 적절한 블록의 개수와 블록 별 쓰레드 개수가 존재함을 알 수 있다.
- 1000x1000 크기의 행렬에서는 블록 개수가 64개일 때 평균적으로 약 2250배의 다른 블록 수에 비해

가장 큰 속도 향상이 있었다.

- 2000x2000 크기의 행렬에서는 블록 개수가 16 개일 때 평균적으로 약 4670 배의 가장 큰 속도 향상이 있었다. 따라서 행렬 사이즈가 커질수록 설정한 블록의 개수가 작아졌을 때. 추가적인 가속화가 발생한 것을 알 수 있다.
- 동일 그룹 내 최장 시간과 비교하여 1000x1000 행렬에서는 약 1.80 배, 2000x2000 행렬에서는 약 2.94 배의 추가적인 가속화를 달성하였다.
- 연구 결과는 학술대회 (한국정보처리학회 춘계학술대회 ASK2024)에 제출되어 2024-05에 발표 예정이다(논문 제목 : Exploration of Optimization Environment for CUDA-based Cholsky Decomposition).

2) 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 시간이 걸리는)에서 벗어나 효율적으로 빠른 진단을 내릴 수 있는 장점이 있다.
- 이러한 ML 기법은 의료 리포트를 학습하여 의사 결정을 내리고, 질병을 자동으로 예측하는 분석 도구이다.
- 본 연구에서는 ML 기법을 이용하여 고성능의 심장 질환 예측 시스템인 HypGB를 개발한다.



[그림 7] HypGB 심장질환 자동 예측 시스템 개요도

Algorithm 1: Pseudocode of Proposed HypGB Algorithm

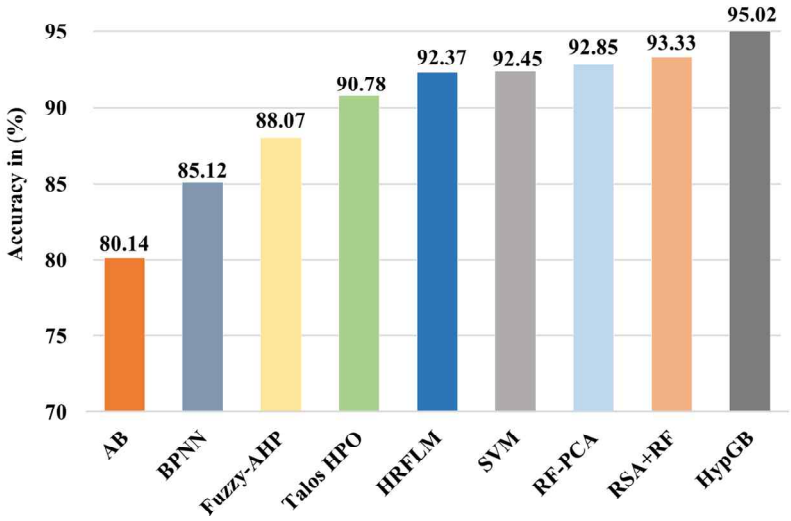
Require: Dataset , Selected features by F by LASSO**,** Search space S, Search algorithm A, Max. number of evaluations **N**

Ensure: Optimized GB Classifier GB_{opt}

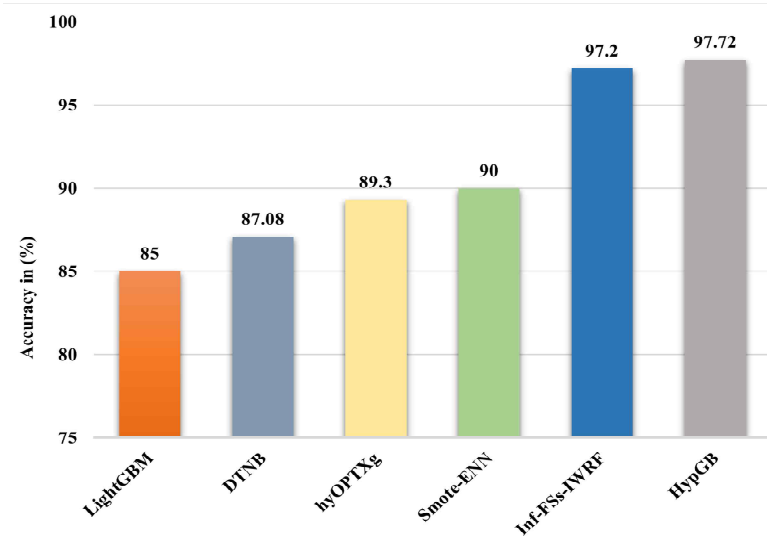
Preprocess $D$and split Dinto D_{train}, D_{test}
Train GBon D_{train}using F
Evaluate the performance of GBon $D_{test}$and get the initial score S_{init}
Initialize history $h$of hyperparameters with the score:h= [ ]
For $i$in range N **do**
    Sample a set of hyperparameters $x$from S using A
    Ifxhas been evaluated before, skip to step 6 **then**
        Train GBmodel on GB_xon D_{train}using $F$and hyperparameters x
        Evaluate the performance of GB_xon $D_{test}$and obtain a score y
        Update the history with the new hyperparameters and score: $h[x] = y$
    end if
end for
Update the ensemble: $F_t(x)=F_{t-1}(x)+\lambda G_t(x)$, where $G_t(x)$is the prediction function of $GB_x$with hyperparameters selected from$h$using the best-performing x
Train GB_{opt}on D_{train}using $F$and x
Evaluate the performance of GB_{opt}on $D_{test}$and obtain a final score S_{final}
If $S_{final}$is better than S_{init}**then**
    returnGB_{opt}
Else
    returnGB
end if

[그림 8] Pseudocode of Proposed HypGB Algorithm

- HypGB는 Gradient Boosting (GB) 모델을 이용하여 임상 데이터에서 심장 질환이 있는 환자들을 분류하고 LASSO feature 선택 기법을 이용하여 가장 이로운 feature들을 찾아낸다.
- GB 모델은 또한 최신의 하이퍼파라미터 최적화 기법인 HyperOpt를 활용하여 최적의 하이퍼파라미터들을 찾아냄으로써 최적화 된다 (HypGB의 개요는 [그림 7]과 [그림 8]의 알고리즘 기술을 참조한다).
- 두 종류의 공개 데이터(Cleveland 심장질환 데이터, Kaggle 심장마비 데이터)를 활용한 실험 결과에 의하면, HypGB는 가장 정확한 feature들과 최적의 하이퍼파라미터 조합을 찾아내어 효율적으로 신장질환자들을 예측한다.
- Cleveland 심장질환 데이터, Kaggle 심장마비 데이터 각각에 대하여 97.32%, 97.72%의 예측 정확도를 보였는데 ([그림 9] 및 [그림 10] 참조), 관련한 이전의 연구들을 뛰어넘는 결과이다. 고성능의 결과로서 HypGB는 의료현장에서 의료진들에게 효율적인 도구로 사용될 가능성을 열었다고 평가된다.
- 연구 결과는 SCI 학술지인 IEEE Access, 2023년 12월호에 게재되었다(제목 : HypGB: High Accuracy GB Classifier for Predicting Heart Disease With HyperOpt H PO Framework and LASSO FS Method).



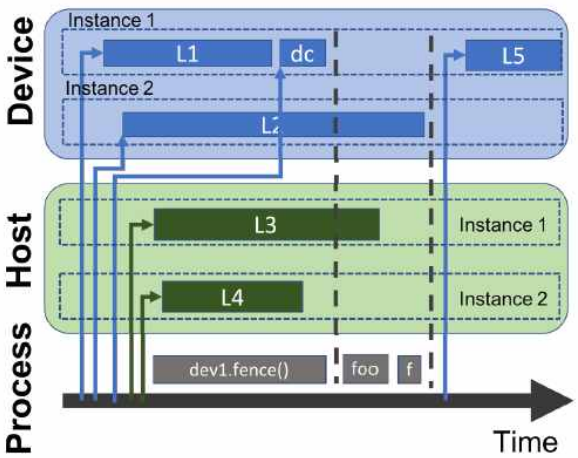
[그림 9] Cleveland 심장질환 데이터를 활용한 실험 결과 (HypGB와 이전 연구의 성능 비교)



[그림 10] Kaggle 심장마비 데이터를 활용한 실험 결과 (HypGB와 이전 연구의 성능 비교)

4. Kkokos 분석

- Kkokos는 C++ 템플릿 기반의 메타 프로그래밍 언어로서, 프로그래머로 하여금 전체적인 일반적인 구조를 작성하도록 하고, 아키텍쳐에 맞는 저수준의 디테일들, 코드 생성 및 성능 최적화는 컴파일러의 작업으로 남긴다.
- 현재 OpenMP. Nvidia CUDA, AMD HIP 등의 각각 다른 아키텍쳐에 타게팅되는 backend들을 제공한다.
- Kokkos의 메모리 모델은 일반 가속기 프로그래밍에서와 마찬가지로 host 메모리, device 메모리가 분리되어 있고, 그들은 공유도 가능한 형태를 띈다. 또한 메모리 모델은 메모리 공간, 레이아웃, 특성을 위한 추상화를 제공한다.
- Kokkos에서 제공하는 주요 병렬 구조는 parallel_for 와 parallel_reduce 이다. 이러한 병렬 구조는 일차원, 다차원, 하이브리드 병렬성을 표현하는데 활용된다. 하이브리드 병렬성은 OpenACC 등과같은 AI에서의 gang, worker, vector 병렬성들의 혼합을 의미한다.
- 그밖에 동기화 구현을 위하여 atomic 연산을 제공한다. +, -, *, % 등의 연산이 integer, float (single-precision, double-precision) 데이터 타입에 대해 제공된다.
- 위와 같은 기능을 제공하는 kokkos로 병렬화된 응용의 성능 최적화는 처음부터 특정 아키텍처를 타겟으로한 API를 사용한 병렬화와 비교하여 많은 차이가 나는 것이 일반적이다. 따라서 이러한 성능 격차를 줄이기 위한 기법의 개발이 필수적이다.
- 가장 먼저 생각할 수 있는 기법은 kokkos backend 컴파일러가 높은 성능의 코드를 생성할 수 있도록, kokkos 코드를 컴파일러 친화적 및 성능 친화적으로 변환하는 것이다. 이를 위하여 특정 응용 프로그램을 분석하여 그에 맞는 프로그램 변환 기법을 적용 또는 개발할 필요가 있다.
- 1차년도 연구에서는 슈퍼컴 5호기의 성능 분석 및 특성화를 위하여 마이크로 벤치마크들을 활용한 스터디를 진행하였는데, 이러한 마이크로 벤치마크들을 앞으로의 kokkos 성능 스터디를 위하여 우선적으로 활용한다.



[그림 11] Kokkos의 multi instance, hybrid 병렬성