M3239.005400 데이터사이언스를 위한 컴퓨팅 2 (001)

Homework #5

Due : 2024/12/01 (Sun)

2024-28413 이정현

1. Matrix Multiplication Single GPU

-병렬화 방식에 대한 설명

최종으로 병렬화된 코드에는 크게 두 가지 방법이 사용되었다.

(1) Tiling (Block단위 연산)

Kernel 안에서 shared local memory를 선언한 후, 행렬들을 작은 블록으로 나누어서 할당해주었다. (BLOCK_SIZE라는 변수 선언 후 블록 크기 조절) Tiling과 shared memory의 조합으로 GPU의 구조를 최대한 효율적으로 사용할 수 있었다. 이에 따라, gridDim은 블록들이 전체 행렬을 커버할 수 있도록 블록 크기에 맞춰서 선언해주었다. 하나의 블록은 행렬의 부분 연산을 담당한다. 또한 local memory에 행렬을 복사할때, 실제 행렬 곱 연산을 수행할 때 총 2번 _syncthreads()를 사용함으로써 thread들이 의도한 실행 순서대로 올바른 결과를 낼 수 있도록 하였다.

(2) 하나의 thread가 여러개의 element 연산 담당

하나의 thread가 원소 하나를 담당하는 것이 아니라 여러개의 원소(4x4 block)들을 담당하도록 하였다. Thread당 원소 개수를 의미하는 (NUM_ELEM_PER_THREAD) NEPT_X, NEPT_Y(각 차원당 한개) 라는 변수를 선언해서 NEPT_X x NEPT_Y 크기의 sub-block을 만들어준 후, 하나의 thread가 이 sub-block안 모든 원소들의 연산을 담당한다.

이 방법은 thread 개수를 줄인다는 것을 포함해서 상당한 장점들이 있다. 첫째, 하나의 thread가 더 많은 연산을 하게 때문에 memory access의 utility를 향상시켜준다. 한번 저장된 데이터의 재사용이 늘어나기 때문에, memory latency의 비용이 전체 성능에 미치는 영향이 상대적으로 줄어드는 효과 또한 볼 수 있다. 그리고 둘째, GPU는 동시에 많은 양의 연산을 병렬 처리할 수 있는 강점이 있기 때문에, 이런 구조는 GPU의 자원을 효율적으로 사용하는 방안이라고 생각할 수 있다.

-성능 최적화를 위한 적용한 방법 및 고려 사항들 논의

1. gridDim과 blockDim의 올바른 값 설정.

최적화 과정에서 tiling 및 single thread-multi element 방법을 적용함에 따라서 그리드 크기와 블록 크기를 적절하게 설정해주는 것이 중요했다. 오류없이, 그리고 모든 크기의 행렬에 대해서 올바르게 작동할 수 있도록 구현한 최종 결과는 다음과 같았다.

```
dim3 blockDim(BLOCK_SIZE / NEPT_X, BLOCK_SIZE / NEPT_Y);
dim3 gridDim((N + BLOCK_SIZE - 1) / BLOCK_SIZE, (M + BLOCK_SIZE - 1) / BLOCK_SIZE);
```

Block 크기에 상관없이 항상 전체 행렬을 커버할 수 있을 만큼의 block들이 만들어지도록 그리드를 잘 형성했고, 하나의 block안에서는 하나의 thread가 연산하는 element개수에 맞춰서 thread들이 생성되도록 구현하였다.

2. BLOCK_SIZE, NEPT_X, NEPT_Y 값을 변화시키면서 최고 성능 탐색.

위 세개의 변수는 이번 과제에서 병렬화 성능의 핵심을 담당하는 변수들이다. GPU resources를 최대한 효율적으로 사용할 수 있는 조합을 찾기 위해서 다양한 숫자들로 성능 최적화를 시도하였다. 그 결과, (64, 4, 4) 일때 가장 좋은 성능을 목격할 수 있었다.

```
#define BLOCK_SIZE 64
#define NEPT_X 4 //number of element per thread X
#define NEPT_Y 4 //number of element per thread Y
```

-matmul_single.cu의 각 부분에 대한 설명

(1) matmul_initialize에서 사용된 API List

-cudaGetDeviceCount : 사용할 수 있는 GPU의 개수를 구한다

-cudaMalloc : GPU에 memory를 할당한다

(2) matmul에서 사용된 API List

-cudaMemcpy : GPU memory에 데이터를 복사해준다

-cudaDeviceSynchronize : 실행중인 GPU kernel들이 모두 완료할때까지 CPU를 멈춰준다

(3) matmul_finalize에서 사용된 API List

-cudaFree : GPU memory를 해제해준다

-장점 (CUDA vs. OpenCL)

-CUDA의 장점 : CUDA는 OpenCL에 비해서 사용하기 쉽고, 성능이 아주 잘 나온다는 장점이 있다. 이번 최적화 방법들은 HW4에서 적용했던 것들과 거의 유사한데, OpenCL에 비해서 호출해야 하는 API의 개수도 훨씬 적고, 성능도 CUDA가 더 잘나오는 것으로 확인되었다. CUDA는 NVIDIA GPU와 더 밀접하게 연관되어있고, GPU를 직접적으로 다룰 수 있기때문에 더 높은 성능을 얻을 수 있다.

-OpenCL의 장점 : OpenCL은 portability(범용성/확장성) 가 높은것이 장점이다. 다양한 GPU뿐만 아니라 다른 가속기들에서도 사용 가능해서 heterogeneous system에서 사용할 수 있는 기회가 많다.

-최적화 방식의 분류 및 각각에 대한 성능 실험 결과

이 부분에서 측정된 성능은 모두 **run.sh에 M=N=K=4096** 옵션을 주고 측정되었다.

- 기본 Skeleton Code 실행결과 **(7 GFLOPS)**
- 2D grid, block 구조 활용하기 **(725 GFLOPS)**

  GPU는 여러개의 block을 동시에 실행할 수 있기 때문에 thread-per-block 한계점을 넘지 않으면서 행렬을 block들로 분할하였다 (block 크기가 16x16일때 최고 성능 목격). 각 block에는 256개의 thread가 존재하고, 하나의 thread는 하나의 element를 계산한다.

```
dim3 blockDim(16,16);
dim3 gridDim((N + blockDim.x - 1) / blockDim.x, (M + blockDim.y - 1) / blockDim.y);
```

```cpp
__global__ void matmul_kernel(float *A, float *B, float *C, int M, int N, int K) {

  int i = blockDim.y * blockIdx.y + threadIdx.y; //row. (.y 가 row!)
  int j = blockDim.x * blockIdx.x + threadIdx.x; //col. (.x 가 column!)

  if(i < M && j < N){
    float sum = 0.0f;
    for(int k=0; k<K; k++){
      sum += A[i * K + k] * B[k * N + j];
    }
    C[i * N + j] = sum;
  }

}
```

또한, kernel안에서 index를 잘 맞춰주었다. block에서의 index에서, (.y) 요소들이 row를 나타내고, (.x) 요소들이 column을 나타낸다는 점에 유의하였다. For문 안에서 매번 업데이트 되고 있었던 C 행렬을 밖으로 뺐고, for문 안에서는 연속적인 메모리 접근과, 계산 값이 sum이라는 변수에 누적되어 저장되게 하였다.

- Kernel 안에서 Shared Memory 활용하기 **(1028 GFLOPS)**

  Kernel이 지속/반복적으로 global memory를 읽고 있기 때문에 메모리 접근 측면에서 매우 비효율적이라고 생각하였다. 따라서, shared memory를 사용하였다. 16x16 크기의 shared memory를 행렬 A, B를 위해 각각 선언해주고, 행렬을 부분저장해서 효율적인 데이터 재사용이 가능하게 하였다. Global memory access를 줄임으로써 큰 성능 향상을 목격할 수 있었다.

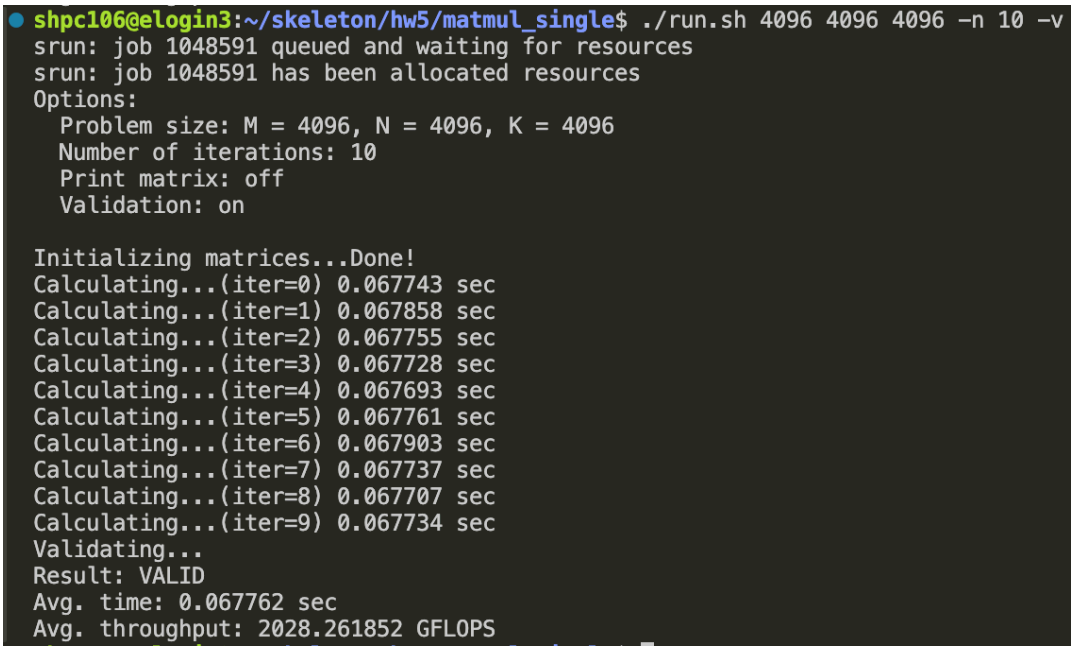
- 하나의 thread가 여러개의 element 계산 **(2028 GFLOPS)**

  Thread가 하나의 원소를 계산하는 것이 아닌, 4x4 크기의 sub-block 계산을 담당하도록 구현하였다. 앞서 다뤘던 것 처럼, thread당 연산 양을 늘려주면 데이터의 재사용이 늘어나고 한 번의 memory access당 더 많은 연산이 일어나기 때문에 전체 성능에서 memory latency의 영향을 상대적으로 감소시켜주는 효과가 있다. GPU의 병렬 연산 특성을 효율적으로 다룰 수 있는 방법이기 때문에 상당한 성능 향상을 목격할 수 있었다. (BLOCK_SIZE = 64)

```cpp
//shared memory
__shared__ float A_block[BLOCK_SIZE][BLOCK_SIZE];
__shared__ float B_block[BLOCK_SIZE][BLOCK_SIZE];

int ty = threadIdx.y;
int tx = threadIdx.x;
int row = (blockIdx.y * blockDim.y + ty) * NEPT_Y; //row. (.y가 row!!)
int col = (blockIdx.x * blockDim.x + tx) * NEPT_X; //col. (.x가 col!!)
```

-성능 : ./run.sh 4096 4096 4096 -n 10 -v 실행결과 2028 GFLOPS



2. Matrix Multiplication Multi GPU

-병렬화 방식에 대한 설명

최종으로 병렬화된 코드에는 이번 과제 1번에서 사용된 tiling, 하나의 thread가 multi element 계산하기 두 가지 방법이 그대로 사용되었고, 이외에도 세 가지 정도의 방법을 추가해서 성능을 최적화하였다. (1), (2)는 앞서 1번 문제에서 작성한 부분과 동일해서 구체적인 설명은 생략하였다.

(1) Tiling (Block단위 연산)

(2) 하나의 thread가 여러개의 element 연산 담당

(3) 행렬 4개의 GPU로 분할하기

각 GPU에 A,C 행렬은 row 기준으로 4분할을 해서 각 부분을 메모리에 저장하였고, B 행렬은 전체를 저장하였다.

(4) 다중 cudaStream 생성 및 비동기 API 호출

4개의 GPU를 효율적으로 사용하기 위해서, 4개의 stream을 생성하고 사용하였다. 하나의 stream에서는 순차적으로 작업이 실행되지만, stream간에는 작업이 동시에(병렬로) 일어날 수 있기 때문에 행렬 곱셈이 4분할되어 동시에 진행될 수 있게 구현하였다. 또한, cudaMemcpyAsync를 이용

해서 host와 device사이의 memory copy를 비동기적으로 수행하도록 하였다. 이 API는 특정 stream과 함께 사용해서 해당 stream의 다른 작업과 겹치지 않도록 data transfer를 관리해준다.

Kernel을 실행할때도 각 GPU마다 고유한 stream을 사용한다. 그리고 kernel 실행 완료를 보장하기 위해서 cudaStreamSynchronize(stream[I])를 이용했고, 이후에 data transfer가 일어나도록 해주었다.

(5) Host에 Pinned Memory 사용하기

CPU와 GPU(device) 사이의 데이터 전송을 효율적으로 만들기 위해서 host memory를 pinned로 만들어주었다. Pinned memory는 운영체제의 virtual memory system에서 paging되지 않는 memory로, 항상 physical memory에 존재하기 때문에 언제든지 접근이 가능하다는 특징을 가지고 있다. 또한, pinned memory는 DMA(Direct Memory Access)를 통해서 GPU와 직접적인 data transfer이 가능하기 때문에 중간 복사 과정 없이 매우 효율적으로 데이터를 직접 전송할 수 있다. 이번 과제에서는 A,B,C 행렬을 pinned memory로 등록하였고, 이후 cudaMemcpyAsync 과정이 더욱 빠르게 일어날 수 있게 되었다.

-성능 최적화를 위한 적용한 방법 및 고려 사항들 논의

1. GPU 4개를 어떻게 사용할 것인지에 대한 결정

다중 GPU를 관리하는 방법은 하나의 thread이용하기, 여러개의 thread이용하기 등 다양한 방법이 있지만, 이번 과제에서는 하나의 thread로 4개의 GPU를 관리하기로 결정하였다. 각 GPU에서 작업이 잘 작동하도록 cudaSetDevice(i)를 적절히 사용하였고, 이 이후 호출되는 동작들은 i번째 GPU에서 실행된다는 사실에 유의해서 코드를 구현하였다.

또한, GPU마다 stream을 생성했으며 cudaMemcpyAsync와 cudaStreamSynchronize 등의 API를 사용해서 각 stream을 관리해주었다.

2. Double buffering

앞서 1번 문제에서 single GPU 최적화를 했을때, single thread-multi element 계산까지 구현한 최적화 방식이 가장 뛰어난 성능을 보여주었다. 이번에도 마찬가지로 그 부분까지 똑같이 구현해서 4개의 GPU를 사용했었고, 2600 GFLOPS를 기록했다. 한 단계 더 최적화를 위해 double buffering 방법을 고려했었다. 데이터의 전송과 연산과정을 겹쳐서 동시에 처리할 수 있게 해주는 최적화 방법인데, 이 방법 적용 결과 2400 GFLOPS로 성능이 소폭 감소하는 것을 목격하였다. 성능 감소에 대해 추론한 이유는 두 가지 정도로, 첫째 이 방법은 버퍼를 두개 유지해야함으로 메모리가 더 많이 필요하고, BLOCK SIZE를 축소할 수 밖에 없었다. 그리고 둘째, 버퍼를 두개 관리하기 위한 logic에서 추가 overhead (ex. Synchronization issue) 등이 발생했을 수 있다. 그리고 무

엇보다 이번 GEMM 코드에서 data transfer이 병목이 아니였다면 double buffering의 효과가 미미했을 수 있겠다고 생각하였다.

3. Pinned memory

따라서, double buffering은 사용하지 않기로 하였고 대신 host memory를 pinned로 사용해야겠다는 생각이 들었다. 이 과정에서 생각할만한 요소가 하나 있었는데, 강의자료나 수업시간에 배웠던 cudaMallocHost API를 사용하려고 했으나 이 API는 새로운 메모리를 할당하고 고정해준다. 그러나 행렬 A,B,C는 이번 과제에서 변경할 수 없는 util.cpp에서 할당되고 있었기 때문에 이 API는 사용하기가 어렵다는 결론에 이르렀다.

따라서 이번 과제에는 cudaHostRegister API를 사용하였다. 이 함수는 이미 존재하는 host memory를 고정해주는 역할을 담당한다. 따라서 수정할 수 있는 코드인 matmul_multi.cu안 matmul() 함수의 시작에 이 API를 호출해서 행렬 A,B,C를 고정 memory로 만들어주고, matmul() 함수의 끝부분에 cudaHostUnregister API를 이용해서 memory를 해제해주었다.

-matmul_multi.cu의 각 부분에 대한 설명

(1) matmul_initialize에서 사용된 API List

-cudaGetDeviceCount : 사용할 수 있는 GPU의 개수를 구한다

-cudaGetDeviceProperties : GPU 정보를 알려준다

-cudaSetDevice : 현재 실행중인 thread가 사용할 GPU device를 지정한다

-cudaMalloc : GPU에 memory를 할당한다

-cudaStreamCreate : GPU stream을 생성한다

(2) matmul에서 사용된 API List

-cudaHostRegister : 존재하는 host memory를 고정 memory로 만들어준다

-cudaSetDevice : 현재 실행중인 thread가 사용할 GPU device를 지정한다

-cudaMemcpyAsync : host와 device 사이의 memory copy를 비동기적으로 수행해준다

-cudaStreamSynchronize : 지정된 stream의 모든 작업이 완료될때까지 CPU 대기시킨다

-cudaHostUnregister : 고정 memory를 해제한다

(3) matmul_finalize에서 사용된 API List

-cudaSetDevice : 현재 실행중인 thread가 사용할 GPU device를 지정한다

-cudaFree : GPU memory를 해제해준다

-cudeStreamDestroy : GPU stream을 해제한다

-최적화 방식의 분류 및 각각에 대한 성능 실험 결과

이 부분에서 측정된 성능은 모두 **run.sh에 M=16384, N=K=4096** 옵션을 주고 측정되었다.

- 기본 Skeleton Code 실행결과 **(29 GFLOPS)**
- 다중 cudaStream 생성 및 cudaMemcpyAsync 사용 **(29 GFLOPS - 변화 X)**

  Stream과 asynchronous memory copy를 사용하는 이유는 data transfer를 연산과 겹치게 하기 위해서이다. Data transfer time이 주된 성능 향상을 방해하는 bottleneck 이었다면 이 시점에서 이 최적화가 많은 성능 향상을 보여주었을 것이다. 그러나 아직 kernel을 최적화하기 전이기 때문에 kernel 실행시간이 전체 실행시간에서 가장 많은 부분을 차지한다. 그러나 앞으로 GPU 4개를 쓰는 부분에 대해 많은 최적화 과정들을 추가할 예정이기 때문에 이 최적화를 미리 적용해주었다.

- 2D grid, block 구조 활용하기 - kernel은 기본 유지 **(330 GFLOPS)**

  GPU는 여러개의 block을 동시에 실행할 수 있기 때문에 thread-per-block 한계점을 넘지 않으면서 행렬을 block들로 분할하였다 (block 크기가 16x16일때 최고 성능 목격). 각 block에는 256개의 thread가 존재하고, 하나의 thread는 하나의 element를 계산한다.

```
dim3 blockDim(16, 16);
dim3 gridDim((N + blockDim.x - 1) / blockDim.x, ((Mend[i] - Mbegin[i]) + blockDim.y - 1) / blockDim.y);
```

- Kernel 간단하게 수정하기 **(1620 GFLOPS)**

  상당히 비효율적으로 계산되고 있었던 kernel에 간단한 수정사항 몇 가지를 적용하였다.

첫째, 행렬의 row, column이 block과 grid의 dimension과 맞도록 수정하였다. row는 y-dimension을 이용해서 계산하고, column은 x-dimension을 이용해서 계산한다.

둘째, 로컬 변수를 선언해서 계산 값이 for loop안에서 C에 바로 업데이트 되는 것이 아닌, loop가 끝나면 로컬 변수에서 C 행렬로 업데이트 되도록 수정하였다. 이를 통해서 메모리의 접근이 for loop안에서 효율적으로 일어날 수 있다. 또한, 인덱스를 올바르게 수정함으로써 하나의 block안 thread들에서 연속적인 메모리 접근이 가능하도록 하였다.

아직도 단순한 kernel이지만, 높은 성능 향상을 목격할 수 있었다.

```
__global__ void matmul_kernel(float *A, float *B, float *C, int M, int N, int K) {

  int i = blockDim.y * blockIdx.y + threadIdx.y; //row
  int j = blockDim.x * blockIdx.x + threadIdx.x; //col

  if(i < M && j < N){
    float sum = 0.0f;
    for(int k=0; k<K; k++){
      sum += A[i * K + k] * B[k * N + j];
    }
    C[i * N + j] = sum;
  }

}
```

- Shared memory 사용 **(1980 GFLOPS)**

  Kernel안에서 지속적으로 global memory를 읽고 있기 때문에 메모리 접근 측면에서 매우 비효율적이라고 생각하였다. 따라서, shared memory를 사용하였다. 16x16 크기의 shared memory를 행렬 A, B를 위해 각각 선언해주고, 행렬을 부분저장해서 효율적인 데이터 재사용이 가능하게 하였다. Global memory access를 줄임으로써 성능 향상을 목격할 수 있었다. Shared memory에 행렬을 저장한 이후와 실제 연산이 끝난 이후 __syncthreads()를 이용해서 thread들이 잘못 실행되지 않도록 동기화를 해주었다.

- 하나의 thread가 여러개의 element 계산 **(2600 GFLOPS)**

  Thread가 하나의 원소를 계산하는 것이 아닌, 4x4 크기의 sub-block 계산을 담당하도록 구현하였다. Thread당 연산 양을 늘려주면 데이터의 재사용이 늘어나고 한번의 memory access당 더 많은 연산이 일어나기 때문에 전체 성능에서 memory latency의 영향을 상대적으로 감소시켜주는 효과가 있다. GPU의 병렬 연산 특성을 효율적으로 다룰 수 있는 방법이기 때문에 상당한 성능 향상을 목격할 수 있었다. (Block size는 64)

- (One thread - multi element) + double buffering **(2400 GFLOPS)**

  Double buffering은 현재 block에 대해 계산하면서 다음 block을 미리 memory에 올리는 과정을 겹치게 함으로써 idle time을 줄이는 최적화 방법이다. (연산과 data transfer 동시에 진행) 연속적으로 연산이 일어날 수 있기 때문에 상황에 따라 효율적인 최적화 방법으로

알려져 있다. 아래 사진과 같이 shared memory를 변경하고, kernel을 변형하였다. (Block size는 32로 진행)

그러나 앞서 살펴본 몇가지 이유들로 오히려 성능이 소폭 감소하는 것을 볼 수 있었고, 최종 최적화된 코드에 적용하지는 않았다.

```
__shared__ float A_block[2][BLOCK_SIZE][BLOCK_SIZE];
__shared__ float B_block[2][BLOCK_SIZE][BLOCK_SIZE];
```

- (One thread - multi element) + Pinned memory **(4365 GFLOPS)**

  가장 성능이 잘 나왔던 kernel로 돌아가서, cudaHostRegister를 사용해서 행렬 A,B,C를 모두 pinned memory로 만들어주었다. 데이터 전송이 매우 효율적으로 변해서 큰 성능변화를 목격할 수 있었다.

```
//register host as pinned memory
CUDA_CALL(cudaHostRegister((void*)A, M * K * sizeof(float), cudaHostRegisterDefault));
CUDA_CALL(cudaHostRegister((void*)B, K * N * sizeof(float), cudaHostRegisterDefault));
CUDA_CALL(cudaHostRegister((void*)C, M * N * sizeof(float), cudaHostRegisterDefault));
```

-성능 : ./run.sh 16384 4096 4096 -n 10 -v 실행결과 4365 GFLOPS

```
shpc106@elogin3:~/skeleton/hw5/matmul_multi$ ./run.sh 16384 4096 4096 -n 10 -v
srun: job 1050685 queued and waiting for resources
srun: job 1050685 has been allocated resources
Options:
  Problem size: M = 16384, N = 4096, K = 4096
  Number of iterations: 10
  Print matrix: off
  Validation: on

Initializing matrices...Done!
Using 4 devices
GPU 0: NVIDIA TITAN RTX
GPU 1: NVIDIA TITAN RTX
GPU 2: NVIDIA TITAN RTX
GPU 3: NVIDIA TITAN RTX
Calculating...(iter=0) 0.123139 sec
Calculating...(iter=1) 0.134267 sec
Calculating...(iter=2) 0.123062 sec
Calculating...(iter=3) 0.122973 sec
Calculating...(iter=4) 0.126483 sec
Calculating...(iter=5) 0.132801 sec
Calculating...(iter=6) 0.123022 sec
Calculating...(iter=7) 0.123116 sec
Calculating...(iter=8) 0.127156 sec
Calculating...(iter=9) 0.123224 sec
Validating...
Result: VALID
Avg. time: 0.125924 sec
Avg. throughput: 4365.765710 GFLOPS
```