SHPC 2021 Fall Term Project Report

2017-17091 이소흔

December 20, 2021

Contents

1 Introduction

본 보고서는 2021학년도 가을학기 확장형 고성능 컴퓨팅 기말 프로젝트 진행과정 및 결과에 대한 내용을 담았다. 2절에서는 프로젝트 진행에 있어서 적용된 성능 향상 기법들에 대해 구현 방식 및 성능 향상의 이유를 논한다. 3장에서는 결과에 대해 간단히 논하고, 4장에서 완성된 프로그램을 어떻게 돌리는지 설명한다.

2 Performance Enhancing Methods

2.1 Basic Implementation

먼저, device 와 host 간의 data transfer 을 최소화하기 위해, input 을 GPU에 보낸 후 모든 flop 연산들을 GPU에서 처리, 이후 다시 host로 보내 output에 다시 저장하는 과정을 반복하였다.

2.2 Naive Parallelization

성능 개선의 방법 중 가장 쉽게 생각할 수 있는 방법은 주어진 kernel 코드에 있는 for문들을 GPU 각 스레드에 분배하여 처리하는 것이다. 예컨대 `N`개의 원소에 대해 `relu`를 계산하는 경우 다음과 같이 병렬화를 할 수 있다.

```
relu<<<N / BLOCK_SIZE, BLOCK_SIZE>>>(args...)
```

`batch_norm`과 `tanh_layer` 역시 동일한 방식으로 처리할 수 있다. `tconv`(Transposed convolution) 같은 경우, 코드에 3겹의 for문이 있는 것을 확인할 수 있으며 `(h_out, w_out, k)`(각각 output의 세로, 가로방향 좌표 및 channel 좌표) 한 쌍에 대해서 하나의 thread를 대응시켜 계산을 진행할 수 있다. `(H_OUT, W_OUT, K)` 크기의 output을 가지는 `tconv`를 병렬화한 코드는 대략 다음과 같다.

```
__global__ void tconv(float *in, float *out, float *weight, float *bias,
                      int H_IN, int W_IN, int C, int K)
    {
        int H_OUT = H_IN * 2, W_OUT = W_IN * 2;

        int h_out = blockDim.x * blockIdx.x + threadIdx.x;
        int w_out = blockDim.y * blockIdx.y + threadIdx.y;
        int k = blockDim.z * blockIdx.z + threadIdx.z;
```

```
        if (h_out >= H_OUT || w_out >= W_OUT || k >= K) return;
    ...
    }

// Execution
dim3 threads(4, 4, 2);
dim3 blocks(H_OUT/4, W_OUT/4, K/2);
tconv<<<blocks, threads>>>(args...);
```

2.2.1 Reducing Divergence

GPU에서는 instruction execution이 warp 단위로 lock-step 방식으로 실행되며, 어떤 if 문 안에 있는 값이 모든 warp 의 모든 thread 들에서 같지 않은 이상 분기되는 모든 instruction을 fetch 하기 때문에 비효율적이다. 따라서, kernel 코드에 있는 if문의 개수를 줄여주는 것으로 속도 향상을 이룰 수 있다.

주어진 transposed convolution 함수 `tconv`에서 `if (h_in % 2 == 0 && w_in % 2 == 0)`에 대한 분기를 없앨 수 있다. `h_in = h_out -3 + r`로 정의되기에 `h_in % 2 == 0` 과 `(r - h_out) % 2 != 0`이 동치이며, 이러한 조건을 if문으로 확인하는 대신`r`에 대한 for문에서 `1 - h_out % 2` 에서 시작, 2 씩 증가하는 값들만 전달하여도 무방하다. `w_in % 2 == 0`조건도 동일한 방식으로 분기 없이 만족시킬 수 있다.

이렇게 분기를 없애는 방법을 통해 `tconv` 함수의 성능이 1.5배정도 향상되었다.

2.3 Transposed Convolution as Matrix Multiplication

성능 개선의 두번째 방법은 transposed convolution을 MM(행렬곱)으로 해석한 뒤, 빠른 행렬 알고리즘을 사용하여 계산을 진행하는 것이다. 먼저 transposed convolution을 어떻게 행렬의 곱으로 나타내는지 확인한 뒤, MM을 빠르게 할 수 있는 방법을 설명할 것이다.

2.3.1 Formulating Transposed Convolution as Matrix Multiplication

Transposed convolution 도 결국 input에 대한 affine transformation 이기 때문에, input $I \in \mathbb{R}^{H_{in} \times W_{in}} \times C$, weight $W \in \mathbb{R}^{5 \times 5 \times K}$ 및 bias $b \in \mathbb{R}^K$ 를 각각 적당히 변형하여 얻은 I', W', b'에 대해 output $O = I'W' + b'$ 로 나타내어질 것이라 기대할 수 있다. 이번 절에서는 이를 직접 찾아볼 것이다.

먼저, output O는 실제로는 1차원으로 저장되어 있으므로, 아래와 같이 정의되는 O'이 실제로

는 O와 대응되는 위치에 같은 값을 가지는 배열임을 알 수 있다.

$$O' = \begin{bmatrix} O[0,0,0] & O[0,0,1] & \cdots & O[0,0,K-1] \\ O[0,1,0] & O[0,1,1] & \cdots & O[0,1,K-1] \\ \vdots & \vdots & \ddots & \vdots \\ O[H_{out}-1, W_{out}-1, 0] & O[H_{out}-1, W_{out}-1, 1] & \cdots & O[H_{out}-1, W_{out}-1, K-1] \end{bmatrix} \tag{1}$$

$$H_{out} = 2H_{in}, W_{out} = 2W_{in} \tag{2}$$

따라서, $O' = I'W' + b'$ 이 되는 I', W', b' 을 찾는다면 정확한 transposed convolution 을 계산할 수 있다.

그 다음, I에 대해 padding 및 stride 를 넣은 행렬을 J라 하자. 구체적으로, I에 대해 하나의 channel에 해당하는 이미지마다 3 left, top zero padding, 2 bottom, right zero padding 및 각 input 사이에 zero padding 을 1개씩 넣은 것을 J라 하자. 예컨대,

$$I = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \Longrightarrow J = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 2 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 & 5 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

와 같이 padding 및 stride 를 고려한 새로운 J를 만들어준다. 이제, output O'가 어떻게 계산되는지를 살펴보면 다음과 같다.

$$O'[0,k] = O[0,0,k] = \sum_{\substack{0 \le r \le 4 \\ 0 \le s \le 4 \\ 0 \le c \le C}} J[r,s,c]W[4-r, 4-s, k, c] + b[k]$$

이를 잘 관찰해보면, Pythonic한 표현을 빌려 다음의 관계들을 기술할 수 있다. (product 는 row-majorized 연산이다.)

$$I^0_{r,s} \coloneqq [J[r,s,c] \texttt{ for } c \texttt{ in range}(C)] \tag{3}$$
$$I'[0,:] \coloneqq \texttt{concat}[I^0_{r,s} \texttt{ for } (r,s) \texttt{ in product}(\texttt{range}(5), \texttt{range}(5))] \tag{4}$$
$$W^k_{r,s} \coloneqq [W[4-r,4-s,k,c] \texttt{ for } c \texttt{ in range}(C)] \tag{5}$$
$$W'[:,k] \coloneqq \texttt{concat}[W^k_{r,s} \texttt{ for } (r,s) \texttt{ in product}(\texttt{range}(5), \texttt{range}(5))] \tag{6}$$
$$O'[0,k] = \langle I'[0,:], W'[:,k] \rangle + b[k] \tag{7}$$

특히 마지막 식 (7)에서 $O'[0,:] = I'[0,:]W' + b$ 임을 알 수 있다. 이제 이를 일반화 시켜서 $O'[d,:], d \neq 0$에 대해서 동일한 관계식을 유도해본다.

$$O'[d,:] = O[\lfloor d/W_{out} \rfloor, (d \mod W_{out}), :] \qquad (\because (1)) \tag{8}$$
$$O[x,y,:] = \sum_{\substack{0 \le r \le 4 \\ 0 \le s \le 4 \\ 0 \le c \le C}} J[x+r, y+s, c] W[4-r, 4-s, k, c] + b[k] \tag{9}$$
$$\tag{10}$$

위의 두 식에서,

$$I^d_{r,s} \coloneqq [J[\lfloor d/W_{out} \rfloor + r, (d \mod W_{out}) + s, c] \texttt{ for } c \texttt{ in range}(C)] \tag{11}$$
$$I'[d,:] \coloneqq \texttt{concat}[I^d_{r,s} \texttt{ for } (r,s) \texttt{ in product}(\texttt{range}(5), \texttt{range}(5))] \tag{12}$$
$$\tag{13}$$

라고 정의하면

$$O'[d,k] = \langle I'[d,:], W'[:,k] \rangle + b[k] \tag{14}$$

즉 $O = O'(\because (1)) = I'W' + b\mathbb{1}$ 임을 얻는다. 이런식으로 transposed convolution을 행렬에 대한 fused-multiply-add 연산으로 나타낼 수 있으며, 이를 어떻게 활용하는지는 2.3.3 에서 다루기로 한다.

2.3.2 Faster Matrix Multiplication

두 행렬 A, B의 곱 AB를 계산할 때, 기존의 tiling 알고리즘에서 사용하였던 것처럼 A, B를 여러개의 tile로 나누어 MM을 진행하는 것은 같으나, tile 의 크기를 다르게 한다. A 의 경우

$T \times T$ tile들로 나누며, B의 경우 $T \times TV$ tile들로 나누어 계산을 진행한다. 이 과정을 그림으로 표현하면 [1]과 같다.

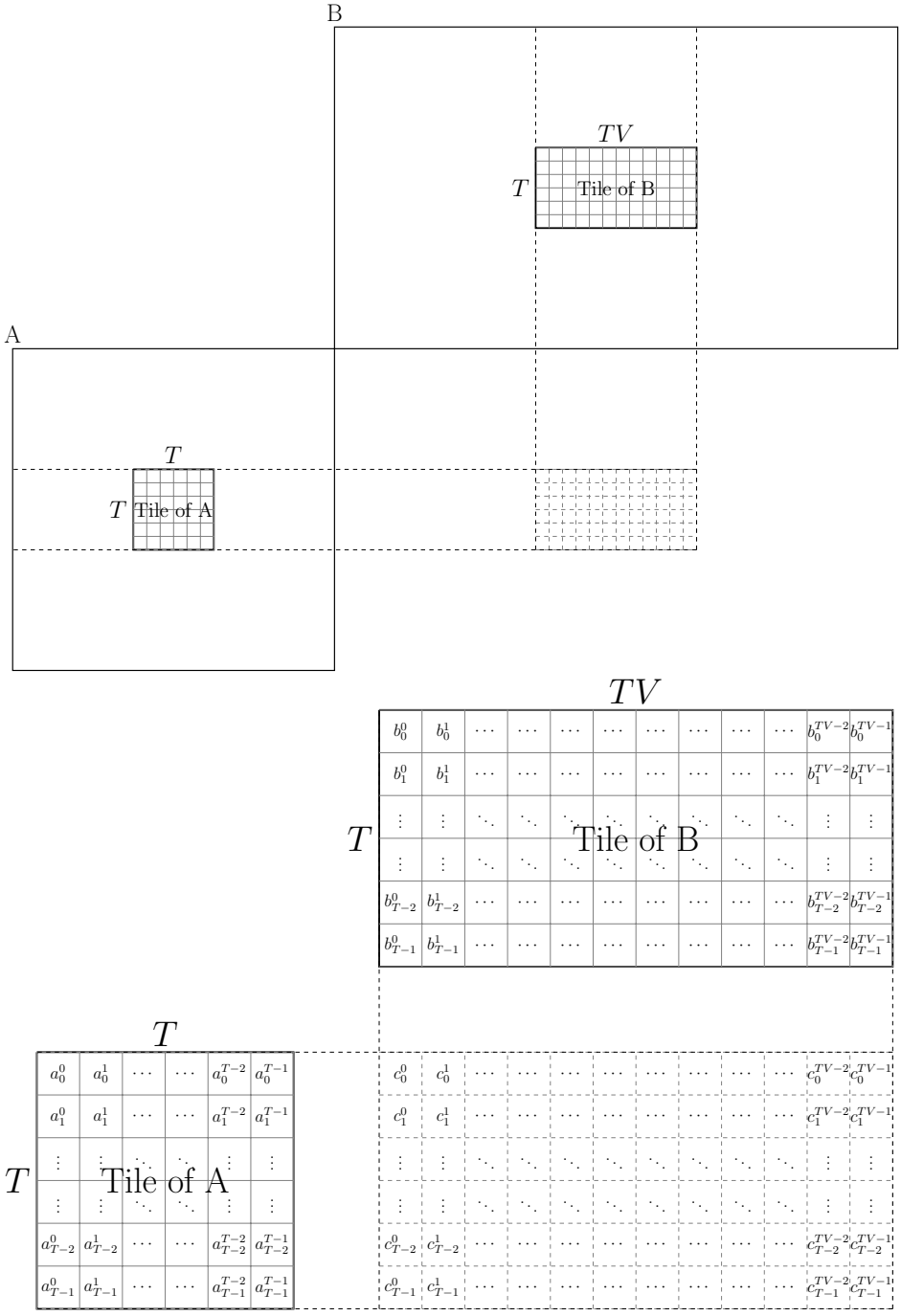

Figure 1: Matrix multiplication, tile-by-tile

해당 그림과 같이 A, B의 타일을 각각 $A_{TILE} = (a_i^j)(0 \le i < T, 0 \le j < T), B_{TILE} = (b_i^j)(0 \le i < T, 0 \le j < TV)$, 그리고 이 둘의 곱을 $C_{TILE} = (c_i^j)(0 \le i < T, 0 \le j < TV)$ 라고 하자. 또한 각각의 i번째 행들을 a_i, b_i, c_i, j번째 열들을 a^j, b^j, c^j와 같이 표현하면, C_{TILE} 의 모든 열들 $c^0, ..., c^{TV-1}$ 에 대해 다음의 식이 성립한다는 것을 알 수 있다.

$$c^j = \sum_{k=0}^{T-1} b_k^j a^k$$

이 성질을 이용하여, C_{TILE} 하나마다 TV개의 thread $0, 1, ..., TV-1$을 배정하여 다음의 알고리즘[1]을 실행한다. 여기서, 첫번째 작업인 shared memory A_{shared} 에 A_{TILE}을 로드하는

Algorithm 1 Faster Matrix Multiplication Algorithm

```
for x = 0, 1, ..., T − 1 do
    for y = 0, 1, ..., T − 1 do
        A_shared[x, y] ← A_TILE[y, x]
    end for
end for
for k = 0, 1, ..., TV − 1 do
    c ← [0] × T
    for t = 0, 1, ..., T do
        b ← B_TILE[t, k]
        c+ = b × A_shared[t, :]
    end for
    C_TILE[:, k] = c
end for
```

작업의 경우 TV개의 thread들이 병렬적으로 수행할 수 있으며, bank conflict 를 피하기 위해서 원래의 tile을 transpose 해서 올린다. 또한 k에 대한 for 문에서 앞서 설명한 C_{TILE}의 계산을 수행하며, TV개의 thread들이 동시에 작업하여도 문제가 없으므로 병렬화가 가능하다.

이 알고리즘이 기존의 tiling 보다 빠른 이유는, 현재 CUDA가 두개의 shared memory 피연산자에 대한 연산을 수행할 수 없기 때문이다. 기존의 tiling 의 경우 Shared memory 에 저장되어있는 두 array `sharedA, sharedB`에 대해 `sharedA[t] * sharedB[k]` 명령을 가장 많이 실행하게 되는데, 이 경우 `sharedA[t]` 혹은 `sharedB[k]` 중 하나를 register 로 올린 뒤 곱셈을 수행하는 두 개의 instruction으로 나뉘어 실행되어 비효율적이다. 따라서, 두 값 중 하나의 값을 register에서 받아오는 경우 성능이 크게 증가한다.

실제 코드에서는 multiplication 에 그치지 않고 fused-multiply-add 와 비슷한 동작을 수행하도록 수정하였으며, 코드 작성에 있어서 다음의 사이트를 참고하였다.[1][2]

2.3.3 Batched Matrix Multiplication

한편, MM은 대체로 피연산자들의 크기가 커질수록 compute-to-memory 비율이 높아져 성능이 향상되므로, 여러개의 MM을 하나의 큰 MM으로 대체할 수 있다면 성능 향상을 기대해볼 수 있다. 따라서, 다음과 같이 2.3.1 에서 소개된 것과 같은 transformed input 여러개($I'_0, ..., I'_{B-1}$)를 이어붙여 하나의 batch로 만든 뒤 transformed weight(W')을 곱한다면 성능이 향상될 것이라 생각하였다.

$$\begin{bmatrix} I'_0 \\ I'_1 \\ \vdots \\ I'_{B-1} \end{bmatrix} W + b\mathbb{1}_{BH_{out}W_{out}} = \begin{bmatrix} I'_0W + b\mathbb{1}_{H_{out}W_{out}} \\ I'_1W + b\mathbb{1}_{H_{out}W_{out}} \\ \vdots \\ I'_{B-1}W + b\mathbb{1}_{H_{out}W_{out}} \end{bmatrix}$$

특히 C 언어는 row-majorized 이기 때문에 위와 같이 각 transformed input들을 세로로 붙이는 과정에서 어떠한 형태의 reshape도 필요 없으며, 계산된 결과 역시 batch로 묶지 않고 계산하는 결과와 정확히 일치하므로 구현에 있어서도 매우 효율적이다. 이를 실제로 구현하였고, 성능이 향상됨을 확인하였다.

2.4 Combining Two Methods

Transposed convolution 에 대해 앞서 소개한 두 가지 최적화 방식(NAIVE, MM(행렬곱)) 중 어떤 것을 적용하는게 더 효율적인지는 input size 와 output channel 수에 따라 다르다. 따라서, DCGAN을 구성하는 각각의 transposed convolution 연산에 대해 최적화 방식을 바꾸어가며 어떤 방식이 더 성능이 좋은지 파악하기 위해 총 세가지 경우: 처음 2개의 transposed convolution 을 MM으로 처리하고 나머지를 NAIVE로 처리한 경우(2 MM + 2 NAIVE), 처음 3개를 MM으로 처리하고 나머지 하나를 NAIVE로 처리한 경우(3 MM + 1 NAIVE), 그리고 모두 MM으로 처리한 경우(4 MM)를 비교해보았으며 이를 Nsight로 확인해본 결과는 [5]와 같다.

Figure 2: 2 MM + 2 NAIVE

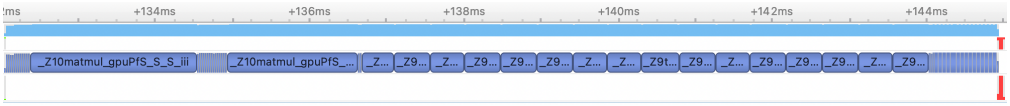

Figure 3: 3 MM + 1 NAIVE

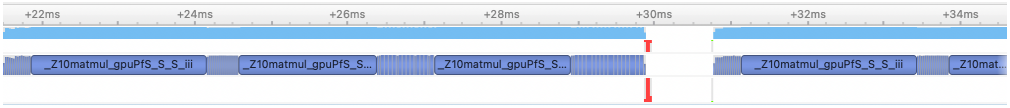

Figure 4: 4 MM

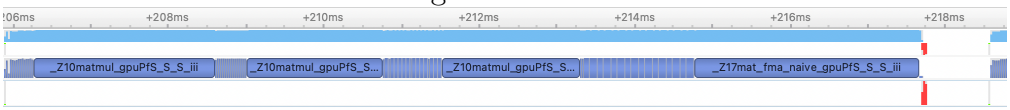

Figure 5: Combining optimization methods: Result on 16 inputs. Time scaled.

보다시피 각 경우의 앞쪽 두개의 MM kernel 까지의 소요시간은 비슷하며 그 이후에 큰 차이가 나는 것을 볼 수 있는데, 2 MM + 2 NAIVE 의 경우 3번째 `tconv`를 NAIVE 방식으로 처리함에 있어서 병목(자그마한 여러개(16개)의 블록들이 naive 한 `tconv` 커널들이다.)이 생기는 것을 볼 수 있고, 4 MM 의 경우 마지막 `tconv`를 MM 으로 처리하는 시간이 길게 걸린다는 것을 알 수 있다. 행렬곱의 경우 행렬의 크기가 커질수록 compute-to-memory 비율이 증가하여 성능이 커지는데, 마지막 channel size 가 3으로 매우 작아 행렬곱의 성능이 크게 낮아져 naive 한 병렬화보다 성능이 떨어짐으로서 이러한 현상이 발생했다고 추측한다.

최적의 성능을 위해 마지막 layer에서만 2.2의 naive한 최적화 방식을 적용하고 나머지 layer 들에서는 2.3의 방식을 적용하였다.

2.5 Pinned Memory and Hiding Memory Operations

2.5.1 Pinned Memory

Pinned memory란, RAM에 고정되어 OS가 강제로 page out 할 수 없는 memory를 의미한다. 반대로 OS가 언제나 page out 할 수 있는 memory를 pageable memory 라고 한다. Host memory를 device로 옮길때 CUDA는 옮기려는 memory가 pinned인지 pageable인지 확인한 뒤 다른 방식으로 memory를 불러온다. 전자의 경우 그냥 physical page만 있으면 불러올 수 있으므로 DMA engine 의 도움을 받지 않고 불러온다. 반면에, 후자의 경우 불러오려는 memory가 paged out 됐을 경우 page fault가 일어날 수 있으므로 CPU, 특히 DMA engine의 도움을 받아야 한다. 즉, page fault cost 및 DMA engine 의 initialization cost 등의 오버헤드가 발생하

므로 pageable memory를 옮기는데 소요되는 시간은 pinned memory를 옮기는 시간보다 훨씬 오래 걸린다.

2.5.2 Detour: Using Pinned Memory without Pinning `outputs`

기본적으로 pageable memory로 선언된 가장 쉬운 해결책은 generator를 통한 계산 결과 값을 저장해야하는 위치인 `outputs`를 pinned memory로 선언하는 것이다. 그러나 `main.c`에 있는 해당 memory 선언을 수정할 수 없다는 조건이 있어, 이를 수정하지 않고 우회적으로 pinned memory를 사용하였다.

이를 위해 먼저

```
cudaMallocHost(&outputs_buffer, BATCH_SIZE * OUTPUT_SIZE * sizeof(float))
```

(`OUTPUT_SIZE` = 64 * 64 * 3) 를 통해 에 batch size 만큼의 output 을 저장할 수 있는 pinned memory `outputs_buffer`를 선언한다. 이후 GPU에서 batch 단위로 input 을 처리한 결과를 일단 `output_buffer` 에 먼저 보낸 뒤 이를 다시 `outputs`에 조금씩 복사하는 방식으로 pinned memory 의 이점을 활용하였다.

다만 `outputs_buffer` 에서 `outputs` memory copy operation은 CPU에서 순차적으로 진행되기에 시간이 꽤 걸리므로(`num_to_gen` 1000 기준 60ms 부근) 이를 GPU kernel 실행 시간에 돌아가도록 하였다. 실제 구현의 경우 host에서 세번째 transposed convolution에 해당하는 kernel 까지 call한 뒤 그 다음 `batch_norm`을 call 하기 전에 해당 memory copy operation 을 실행하였으며 이에 대한 Nsight 결과는 다음[6]과 같다.

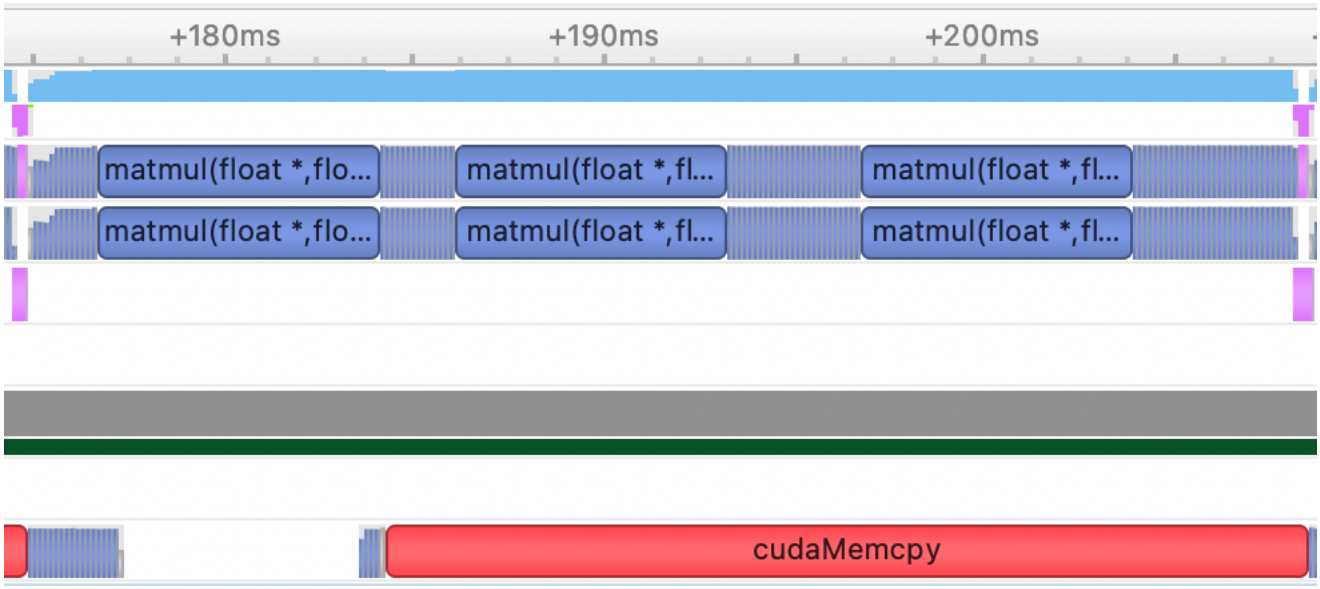

Figure 6: Host memory copy operation hided by kernel operations

보다시피 맨 아래 칸에 기록된 것들이 host 에서 부른 CUDA API 들인데, 도중에 kernel call 을 멈춘 구간이 있는 것을 알 수 있다. 이 구간에서 `outputs_buffer`에 있는 출력값을 `outputs` 에 옮겼을 것이며, 이는 kernel execution 에 완벽히 가려진다는 것을 확인할 수 있다.

2.6 Utilizing More GPUs

마지막으로, 사용 가능한 두개의 노드를 모두 활용함과 동시에 각 노드에 연결되어있는 4개의 GPU를 모두 활용하기 위해 MPI 를 사용한 병렬화를 진행하였다. `MPI_Get_processor_name` 함수를 통해 확인해본 결과, 2개 노드마다 process를 n개씩 launch 하는 경우 첫번째 노드에 rank $0, \cdots, n-1$, 두번째 노드에 rank $n, \cdots, 2n-1$ 이 배정된다는 규칙을 발견하였고 이를 사용하여 각 mpi process가 `mpi_rank % (mpi_size / 2)` 번째 GPU를 사용하도록 설정하였다. 이에 따라 각 rank 마다 어떤 노드에 배치되며 해당 노드의 몇번째 GPU를 쓰는지는 다음과 같은 표로 주어진다.

Table 1: `--n-tasks-per-node=1`

	GPU 0
node 0	rank 0
node 1	rank 1

Table 2: `--n-tasks-per-node=2`

	GPU 0	GPU 1
node 0	rank 0	rank 1
node 1	rank 2	rank 3

Table 3: `--n-tasks-per-node=4`

	GPU 0	GPU 1	GPU 2	GPU 3
node 0	rank 0	rank 1	rank 2	rank 3
node 1	rank 4	rank 5	rank 6	rank 7

3 Result

2.2.에서 `tanh, batch_norm, relu` kernel 을 실행할 때 한 block 당 thread 개수를 `BLOCK_SIZE`, 2.3.2에서 나온 T, V 값을 각각 `T_SIZE, V_SIZE`, 2.3.3 에서 언급한 batch 당 input 의 개수를 `BATCH_SIZE`라고 할 때 `BLOCK_SIZE=128, T_SIZE=16, V_SIZE=4, BATCH_SIZE=100` 하에서 노드 당 GPU 사용 개수 및 `num_to_gen`(input 개수)에 따른 실행 시간은 다음과 같았다.

`num_to_gen`	GPU=1	GPU=2	GPU=4
10	0.039	0.088	0.154
100	0.065	0.102	0.155
880	0.248	0.197	0.224
1000	0.278	0.224	0.236
2224	0.485	0.362	0.311
2512	0.536	0.391	0.333
4856	0.955	0.603	0.506
5248	1.014	0.647	0.504

Table 4: Performance(sec) by `num_to_gen`, # of GPU per node

num_to_gen이 작은 경우에는, GPU 개수가 적을 때 오히려 성능이 더 좋은 것으로 나타난다. 이는 계산량이 적기에, GPU 당 처리해야하는 계산량 감소에 따른 성능 증가보다 mpi process 수 증가에 따른 통신 비용 증가 등의 오버헤드 때문에 발생하는 성능 악화 효과가 더 크기 때문이라고 생각한다. num_to_gen이 점차 커지면서 노드 당 사용하는 GPU 개수가 많을수록 성능이 더 좋아지는 경향이 강화되어, 성능의 격차가 점차 벌어짐을 확인할 수 있다.

4 Execution Guideline

- 기본 실행법은 다음과 같다.

```
make run_parallel INPUT=(input filename without .txt)
OUTPUT=(output filename without .txt/.bmp) NGPUS=(#
of GPUs per node)
```

  - 예시 : `make run_parallel INPUT=input1 OUTPUT=output1 NGPUS=4`
    ⇒ `input1.txt`를 입력받아 `output1.txt, output1.bmp`를 출력하며, 이 과정에서 노드별로 GPU를 4개 사용한다.
  - `NGPUS` 로 가능한 값은 1,2,4이다.

- `num_to_gen` 이 200 이하인 경우 `NGPUS=1`, 200 이상 1000 이하인 경우 `NGPUS=2`, 1000 이상인 경우 `NGPUS=4`가 효율적이다. 선택하기 어려운 경우에는 대체로 `NGPUS=4`로 두는 것이 효율적이다.

- 방금 말씀드린 것처럼, `num_to_gen` 이 1000 보다 작은 경우는 NGPUS=2 로 돌려주시길 부탁드립니다!

References

[1] URL: https://github.com/Huanghongru/SGEMM-Implementation-and-Optimization.

[2] URL: https://ecatue.gitlab.io/gpu2018/pages/Cookbook/matrix_multiplication_cuda.html.