SHPC 2021 Fall Term Project Report

2017-17091 Soheun Yi

October 3, 2022

Contents

1 Introduction

This report contains process and result of the term project of Scalable High Performance Computing(SHPC), Fall 2021. The goal of this project is to perform generation phase of the GAN as fast as possible in limited resources. This is how the report is organized : in Section 2, we propose methods for enhancing performance of transposed convolution and explain how those contribute to performance. In following Section 3 and 4, we introduce results of proposed methods and give guidelines for executing the implemented program.

2 Performance Enhancing Methods

2.1 Basic Implementation

First of all, we have repeated process of sending inputs to GPUs and perform FLOPs there, and retrieving it back to the host and save in output variables, for the sake of time efficiency regarding data transfer.

2.2 Naive Parallelization

The most naive way of improving performance is to distribute for-loops in kernel codes to multiple threads in GPUs. For instance, we can reduce time spent for calculations regarding `relu` for `N` elements by implementing parallelization as the following code:

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

`batch_norm` and `tanh_layer` operations can be processed with the same procedure. For `tconv`(Transposed convolution) operation, we can see that there are three hierarchial for-loops and we can match a thread for each `(h_out, w_out, k)`(vertical and horizontal coordinate of an output, repsectively.) pair to parallelize this operation. The following is implementation of aforementioned parallelization for an output of size `(H_OUT, W_OUT, K)`:

```
__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

IN GPUs, instruction executions are held in units of warps following the lock-step procedure. Therefore, if-branches are inefficient because all single instructions generated within if-branches are fetched, as long as values dealt in all threads and warps do not coincide. This implies we can enhance efficiency by reducing if-branches in the kernel code.

We can observe that we can remove divergence regarding `if (h_in % 2 == 0 && w_in % 2 == 0)` written in `tconv`. Rather than utilizing if-branches, we can manually increment values by 2 and pass them to `r` by modifying the for-loop on r without changing the behavior of the code. This have brought about 1.5× improvement in terms of calculation time.

2.3 Transposed Convolution as Matrix Multiplication

The second possible enhancement strategy is to view transposed convolution as a matrix multiplication(MM) and implementing a powerful MM algorithm. We first discuss how we can represent transposed convolution as a MM, and introduce a powerful MM algorithm.

2.3.1 Formulating Transposed Convolution as Matrix Multiplication

As transposed convolution is an affine transformation of an input, we can reasonably expect the output O is represented as $O = I'W' + b'$, where I', W', b' are adequate transformations of input $I \in \mathbb{R}^{H_{in} \times W_{in}} \times C$, weight $W \in \mathbb{R}^{5 \times 5 \times K}$, and bias $b \in \mathbb{R}^K$.

As (three-dimensional) output O is saved as one-dimensional array, O' as the following representation is the two-dimensional array which corresponds O.

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

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

We now aim to find I', W', b' which satisfies $O' = I'W' + b'$ to find an accurate formula of transposed convolution.

Denote J is a matrix which has applied padding(left=3, right=2, top=0, bottom=2) and stride(=1) on I. As an example,

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

Now, observe how O' is calculated. We can see that

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

Using `Python`-ic notations(product is a row-majorized operation.),

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

The last equation (7) implies $O'[0,:] = I'[0,:]W' + b$. We can generalize this equation to achieve a formula for $O'[d,:]$.

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

With defining

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

we have

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

or equivalently $O = O'(\because (1)) = I'W' + b\mathbb{1}$. In this way, we can formulate transposed convolution as a fused-multiply-add operation on matrices. We discuss how we can use this relation in 2.3.3.

2.3.2 Faster Matrix Multiplication

For a faster MM algorithm than the simplest tiling MM algorithm, we divide operands into multiple tiles analogous to the simplest algorithm but differ in size of tiles. In calculating

multiplication of two matrices A and B, we divide A into $T \times T$ tiles and B into $T \times TV$ tiles, where T and V is size of a tile and a constant, respectively. Refer to Figure 1 for a depiction for this procedure.

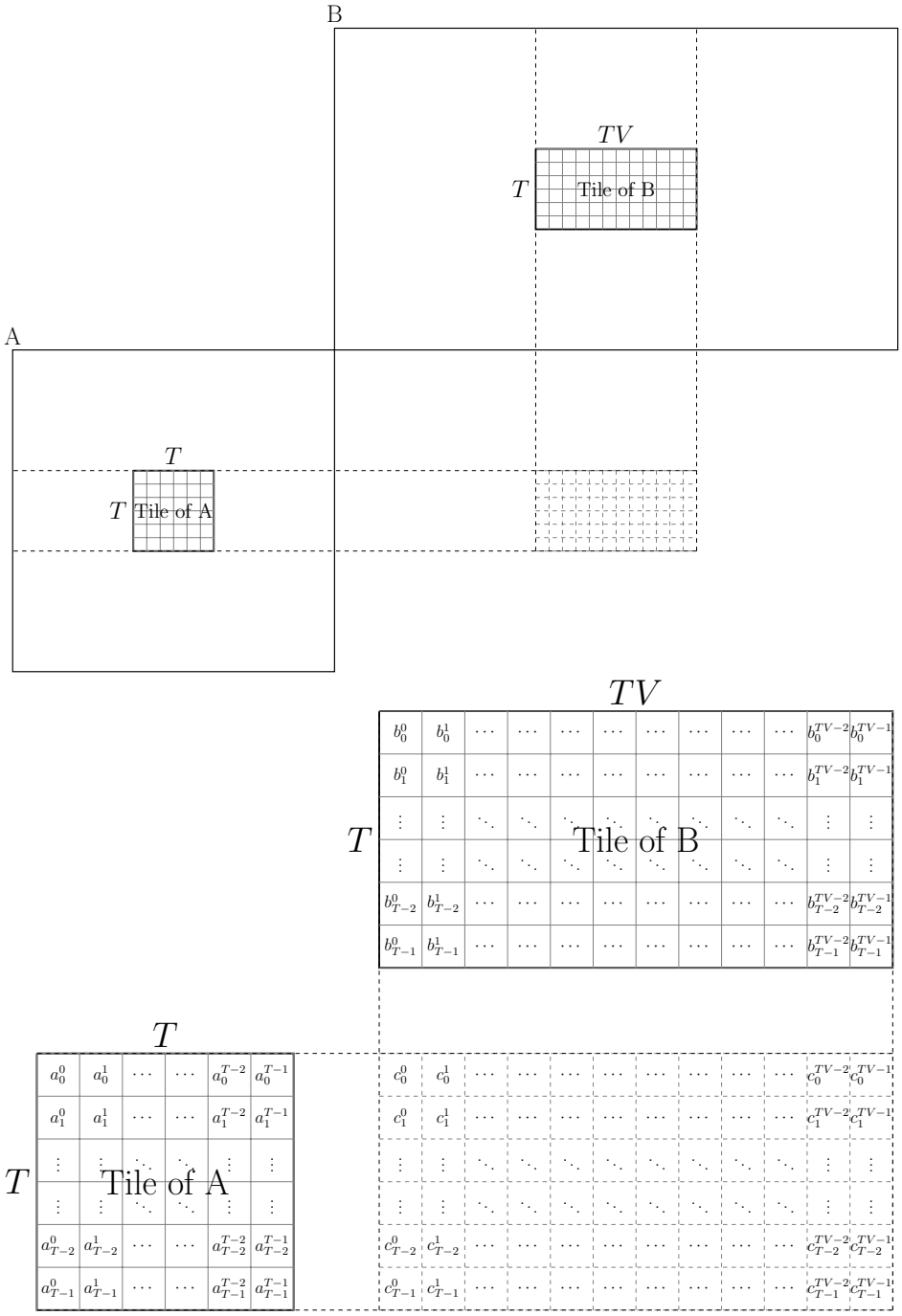

Figure 1: Matrix multiplication, tile-by-tile

As in Figure 1, denote a tile of A by $A_{TILE} = (a_i^j)(0 \leq i < T, 0 \leq j < T)$ and a tile of B by $B_{TILE} = (b_i^j)(0 \leq i < T, 0 \leq j < TV)$. Similarly, denote their multiplication, $C_{TILE} = A_{TILE}B_{TILE}$ by $C_{TILE} = (c_i^j)(0 \leq i < T, 0 \leq j < TV)$. Let i'th rows of them as a_i, b_i, and c_i respectively and j'th columns as a^j, b^j, and c^j respectively. Then, we have

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

Utilizing this property, we employ TV threads to run the following Algorithm 1.

Algorithm 1 Faster Matrix Multiplication Algorithm

for $x = 0, 1, ..., T-1$ **do**
  for $y = 0, 1, ..., T-1$ **do**
    $A_{shared}[x, y] \leftarrow A_{TILE}[y, x]$
  end for
end for
for $k = 0, 1, ..., TV-1$ **do**
  $c \leftarrow [0] \times T$
  for $t = 0, 1, ..., T$ **do**
    $b \leftarrow B_{TILE}[t, k]$
    $c+ = b \times A_{shared}[t, :]$
  end for
  $C_{TILE}[:, k] = c$
end for

For loading A_{TILE} on a shared memory A_{shared}, we can use TV threads concurrently. To avoid bank conflicts, we first transpose original tiles and then load them. We can successfully parallelize the algorithm by this way.

This algorithm is faster than the original tiling algorithm because current CUDA cannot perform an operation dealing with operands from two different shared memories. The original algorithm mostly performs `sharedA[t] * sharedB[k]` for two different shared memories `sharedA, sharedB`, which is extremely inefficient as two separate instructions are executed: loading one of `sharedA[t]` or `sharedB[k]` on a register, and then performing float multiplication.

For better performance, we implemented similar operation to fused-multiply-add operation referrring to [1], [2].

2.3.3 Batched Matrix Multiplication

Efficiency of MM gets better as size of operands get bigger, since compute-to-memory becomes more efficient. Therefore, we can expect improvement in performance if we replace multiple MMs into a single, larger MM. To obtain this, we make a batch and use it as an operand by concatenating transformed inputs $I'_0, \ldots, I'_{B-1}$ and then multiply transformed weight W', as explained in 2.3.1.

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

Especially, `C` language is row-majorized, so we do not need any form of reshaping while concatenating transformed inputs horizontally, and at the same time we do not need to take care of recovering evaluations of calculations. We have implemented this and confirmed improvment in performance.

2.4 Combining Two Methods

Whether which of the two optimization methods introduced(NAIVE, MM) is better depends on the input size and number of output channels. Therefore, we have considered three cases: first, preceding two transposed convolutions are done with MM and others with NAIVE(2 MM + 2 NAIVE), second, preceding three transposed convolutions are done with MM and the other with NAIVE(3 MM + 1 NAIVE), and lastly all transposed convolutions are done with MM(4 MM). We investigated each cases using Nsight and we have obtained results in Figure 5.

Figure 2: 2 MM + 2 NAIVE

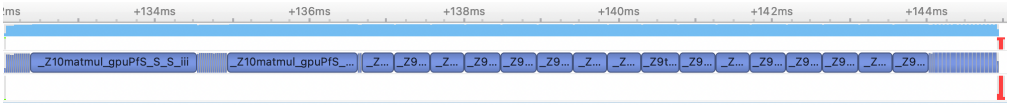

Figure 3: 3 MM + 1 NAIVE

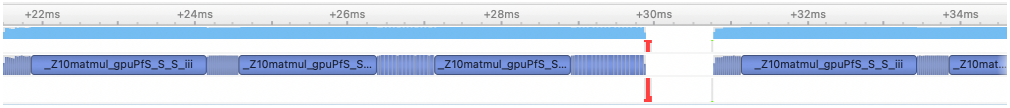

Figure 4: 4 MM

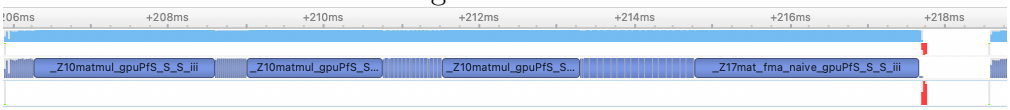

Figure 5: Combining optimization methods: Result on 16 inputs. Time scaled.

As one can see, first two MM kernels take similar amount of time, but other operations differ largely. In the case of 2 MM + 2 NAIVE, bottleneck in third `tconv` appears. On the other hand, last `tconv` in the case of 4 MM takes long time as channel size(=3) of the last operation is relatively small to reduce compute-to-memory and thereby aggravates performance to be lower than NAIVE operation. Therefore, we have applied 3 MM + 1 NAIVE to obtain the best performance.

2.5 Pinned Memory and Hiding Memory Operations

2.5.1 Pinned Memory

Pinned memory refers to memory fixed to the RAM and thereby cannot be paged out by the OS. The counterpart term is named pageable memory. In transferring host memory to device memory, CUDA checks if such memory is either pinned or pageable. If the memory is pinned, CUDA only needs physical page of the memory, and therefore does not require aid of DMA engine. On the other hand, CUDA needs DMA engine for responding correctly to page fault. This causes certain overhead on loading memory, resulting a longer time spent for loading pageable memory than that or pinned memory.

2.5.2 Detour: Using Pinned Memory without Pinning `outputs`

We took a detour to use pinned memory as there is a restriction that we cannot modify `main.c`. We first declared

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

$$\texttt{OUTPUT\_SIZE} = 64 * 64 * 3)$$

to save batched output on a pinned memory `outputs_buffer`. Thereafter, we processed inputs by batch and sent it to `output_buffer`, and then gradually copying it onto `outputs`. Meanwhile, copying values from `outputs_buffer` to `outputs` are performed sequentially thus takes some time(about 60ms for `num_to_gen` = 1000) so we overlapped(hided) this time with GPU kernel runtime. Regarding implementation, we have started this copying operation after the third `tconv` operation and before next `batch_norm` call. The result for this is as Figure 6

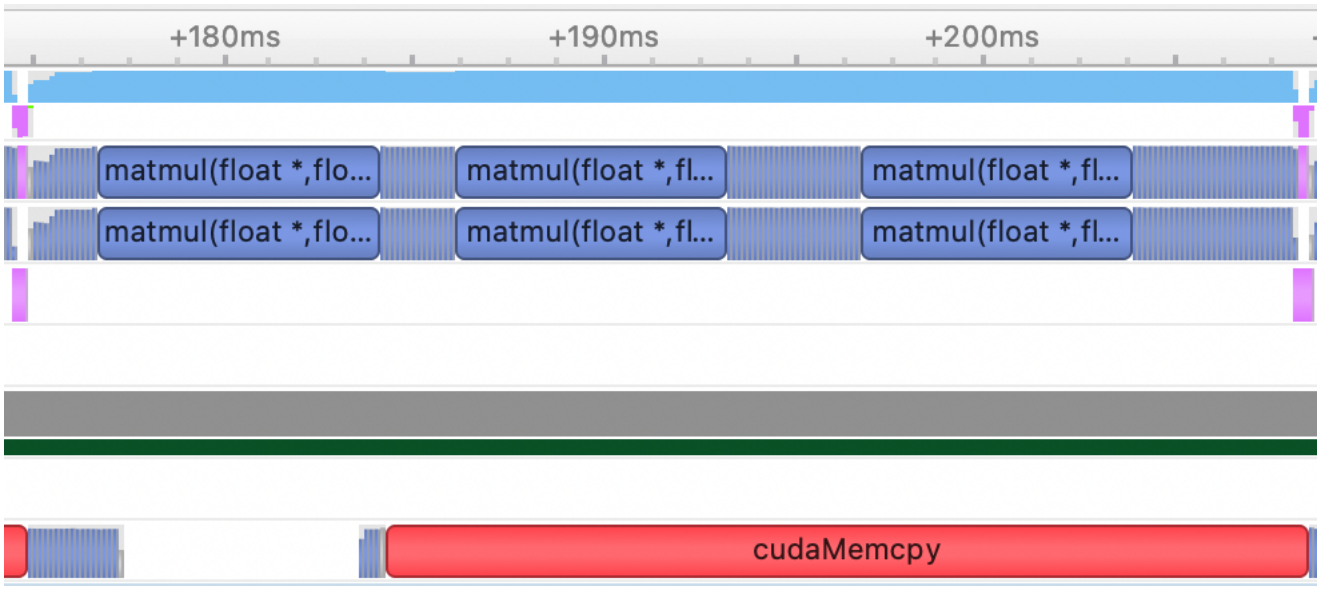

Figure 6: Host memory copy operation hided by kernel operations

As one can see, the ones at the bottom are CUDA APIs called from the host, and there are some stall of kernel calls. Such interval correpsonds to the memory copying operation, which is perfectly hided behind kernel executions.

2.6 Utilizing More GPUs

Lastly, we have used all available two nodes, each with four GPUs. In deploying n processes for each nodes, we have found that mpi process ranks for the first node are $0, \cdots, n-1$ and $n, \cdots, 2n-1$ for the other. Using this pattern, we made each mpi process to use `mpi_rank % (mpi_size / 2)`'th GPU which can be marshalled as the following:

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

Denote `BLOCK_SIZE` as a number of blocks per threads regarding `tanh, batch_norm, relu` kernels, `T_SIZE, V_SIZE` as values of T, V introduced in 2.3.2 respectively, and `BATCH_SIZE` as inputs per batches in 2.3.1. Configuring `BLOCK_SIZE=128, T_SIZE=16, V_SIZE=4, BATCH_SIZE=100`, running time of the whole operation was as Table 4.

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

When `num_to_gen` is small, it reveals that the performance is better with less GPUs. This is allegedly due to relatively large overhead caused by transaction between mpi processes than cost reduced by less calculation per GPUs. As `num_to_gen` gets larger, utilizing more GPUs achieve far higher performance, due to reduction of calculations per GPUs.

4 Execution Guideline

- The basic execution is as follows.

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

  - E.g. : `make run_parallel INPUT=input1 OUTPUT=output1 NGPUS=4`
    ⇒ Get `input1.txt` as an input, print `output1.txt, output1.bmp` as an output with using 4 GPUs per nodes.
  - Possible values for `NGPUS` are 1,2, and 4.

- If `num_to_gen` is less than 200, define `NGPUS=1`. If it is between 200 and 1000, define `NGPUS=2`. Otherwise, define `NGPUS=4`. These are optimal choices. If one cannot judge which one is optimal, take `NGPUS=4`, which is generally the best option.

References

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

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