Assignment A3: Compiling DNNs with TVM

1st William Daugherty-Miller
M Eng. Operations Research and Information Engineering
Cornell Tech
New York City, United States
wld37@cornell.edu

***Abstract*—The speedup calculations found in this assignment highlight the effectiveness of optimizing deep learning models with TVM (Tensor Virtual Machine). For all cases, the optimized runtimes are significantly lower than the unoptimized runtimes, indicating that TVM can improve model performance on various devices. In particular, the speedup factor is higher for Colab, which is a cloud-based platform, compared to the G2 instance, which is a local server. This highlights the importance of optimizing models for cloud-based platforms where resources are shared among many users.**

BACKGROUND

In this assignment, we will use TVM to optimize DNN primitives on CPU, GPU and FPGA devices. TVM is a domain-specific compiler for machine learning in which the program computation specification and schedule are decoupled. This means that we specify the computation separately from the optimizations that actually allow the code to run efficiently on hardware.

You will optimize the following primitives:

- 1D convolution on CPU†.
- 1D convolution on GPU.
- GEMM on GPU†.
- 2D Depthwise Separable Convolution on GPU.

† A Baseline implementation is provided for the 1D convolution on the CPU, and GEMM on GPU; you are required to optimize these implementations. For other problems, you are required to write the baseline implementation and optimize them.

For each of those primitives, there is a functionality test that will run to ensure that your optimized code is still functional. Additionally, we will benchmark all submissions to find the fastest solution. Higher marks will be awarded to faster solutions!

You are expected to perform at least three distinct optimizations for each operation, but you are encouraged to do more. There are plenty of examples on the TVM website to guide you on the kind of optimizations that work on each device. For example, this GEMM on CPU optimization showcases 6 distinct optimizations. It will be very useful for you to walk through these optimizations and understand them before diving into your solution for the questions above. Other useful examples include this 2D Conv Optimization on GPU and GEMM on FPGA (VTA).

You will use TVM's schedule primitives to perform the optimizations. You are not allowed to use AutoTVM, although you may do so for comparison [1 mark extra credit].

In your code submission, make sure to include all of your optimizations, ideally, organized into different functions.

ASSIGNMENT

- Your report should include tables or charts to showcase the runtime and speedup (relative to unoptimized TVM implementation) of each DNN primitive. For the CPU/GPU it would also be useful to include the speedup/slowdown relative to the provided torch/numpy reference.
- For each distinct optimization that you performed, please explain it in some detail: why you performed this optimization? And what impact does it have on the hardware execution? You are encouraged to use figures for explanation, similar to the examples on the TVM website. It may also be useful to show the effect on the TVM IR from the tvm.lower function as this highlights the effect of optimizations–pseudo-code (see Figure 5.8 in textbook) also works well for this purpose.
- You may find it useful to plot space-time diagrams of your memory accesses (see Figure 5.9 in our textbook). These may aid your choice of tiling and vectorization optimizations. It is not required to include these plots in your report but you may choose to do so if it helps. Note: If you tried an interesting optimization but it did not improve your runtime, you can also describe it in your writeup and report on the slowdown and potential reasons why it didn't work. You will be awarded partial marks for this if you provide a meaningful analysis.
- Please do not include your source code in the report unless you are referencing it in the text–your code will be assessed separately through your Github submission.

1D CONVOLUTION ON CPU

A. *Unoptimized Code*

The code defines a function, make_conv1d_cpu_scheduler, that creates a CPU schedule for a 1D convolution operation. The function takes two parameters: M and N, which represent the sizes of the input and filter tensors, respectively. It creates two placeholders using the TensorFlow/TVM API to represent the input and filter tensors. It then defines a reduce axis with a range of (0, M + N - 1) to perform the convolution operation. A compute operation is defined to calculate the convolution of A and W. The schedule object is created using te . create_schedule (B.op) and returned along with the input and filter tensors, and the computed output tensor.

B. Optimizations

We performed the following optimizations on the 1D convolution on CPU:

1) Loop splitting: x_out, x_in = s[B]. split (B.op. axis [0], factor=factor) and k_oout, k_in = s[B]. split (k, factor=factor) are splitting the output loop and reduction loop, respectively. The loop splitting optimization divides the loops into two nested loops. The outer loop iterates with a larger stride (controlled by the factor), while the inner loop iterates over smaller chunks of data. This optimization helps to improve cache locality and better utilize the SIMD capabilities of the hardware.
2) Loop reordering: s[B]. reorder (x_out, k_oout, k_in, x_in) changes the order of loop execution. Reordering the loops ensures that the computation proceeds in a cache-friendly manner by accessing elements in a sequential pattern, reducing cache misses and improving overall performance.
3) Loop unrolling: s[B]. unroll (k_in) unrolls the inner reduction loop. Unrolling a loop means replicating the loop body multiple times, reducing the number of loop iterations. Loop unrolling can improve performance by reducing loop overhead and exposing more opportunities for instruction-level parallelism.
4) Vectorization: s[B]. vectorize (x_in) vectorizes the inner output loop. Vectorization involves transforming scalar operations into vector operations that can be executed in parallel using SIMD hardware instructions. This optimization can significantly speed up the execution of the code on hardware with SIMD capabilities.
5) Caching:s[s.cache_read(A_Pad, "local", [B])]. compute_at(s[B], k_oout) and s[s.cache_read(W, "local", [B])]. compute_at(s[B], k_oout) cache the intermediate results of A_Pad and W in local memory. Caching intermediate results in faster memory can help reduce memory latency and improve performance by reducing the need to fetch data from slower global memory during computation.

C. Failed Optimizations

Fusing was attempted, but for this specific code, it might not be beneficial for performance due to potential interference with other optimizations, worse cache locality, and limited opportunities for parallelism. Loop fusion effects can be problem and hardware-dependent, so it's best to experiment with different combinations of optimizations and measure performance on the target hardware to determine the most effective set of optimizations.

D. AutoTVM

To apply the optimized schedule found by AutoTVM to the make_conv1d_cpu_scheduler function, the function must be updated to accept a configuration argument. The configuration specifies the best tile sizes and optimization passes that were found during tuning. The schedule optimization passes are applied to the function using the configuration, and the optimized function is built and executed to measure performance. The resulting output can be compared to the expected output to validate correctness. This process allows AutoTVM to efficiently optimize the schedule of the function for the CPU, reducing runtime and improving performance.

E. Comparison

Given observed values listed in the table below, we can calculate the speedup factor.

Device	Unoptimized	Optimized	Autotvm
Colab	0.030453567	0.00028571	0.000795371
G2	0.68330	0.12174	NA

TABLE I: Run-time in seconds for CPU Scheduler

If we define the speedup factor as the following: Speedup factor = Unoptimized runtime / Optimized runtime

For Colab:

$$\text{Speedup factor} = \frac{0.030453567}{0.00028571} = 106.71$$

For G2:

$$\text{Speedup factor} = \frac{0.68330}{0.12174} = 5.61$$

To calculate the speedup factor from unoptimized to Autotvm, we can use the following formula:

Speedup factor = Unoptimized runtime / Autotvm runtime

For Colab:

$$\text{Speedup factor} = \frac{0.030453567}{0.000795371} = 38.33$$

We note that AutoTVM was not included on the leaderboard to avoid confusion and not count it as an actual submission.

F. Space-Time Graphs

This graph I believe to be incorrect but it and others like it throughout the report were an attempt to expose myself to creating this type of plot and learning what kind of educated assumptions could be made from the tvm.lower() function.

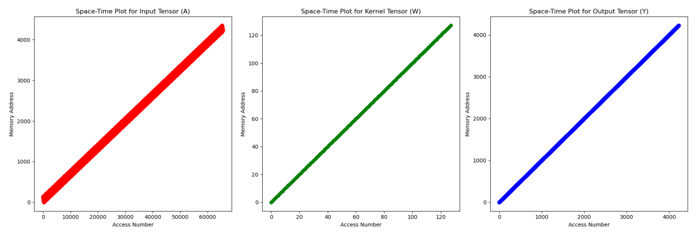

Fig. 1: Space-Time Diagram

1D CONVOLUTION ON GPU

G. Unoptimized Code

The implementation of make_conv1d_gpu_scheduler function performs a 1D convolution operation between two 1D arrays, A and W. It first creates placeholders for the input arrays A and W, and then pads the array A with zeros on both sides to create the array Apad. A reduction axis, k, is defined for the summation during the convolution operation. The result of the convolution, B, is computed using a lambda function that calculates the sum of element-wise products between the padded array Apad and the kernel W. The function then creates a schedule for this operation, which is used later to optimize the performance when running on a GPU. The schedule includes parallelizing over the outer axis of Apad and B, vectorizing over the inner loop of B, binding the outer and inner axes of B and Apad to GPU threads, and setting a pragma for unrolling. The function returns the created schedule and the input and output tensors A, W, and B.

H. Optimizations

We performed the following optimizations on the 1D convolution on GPU:

1) Padding the input tensor (Apad):
   - This optimization adds padding to the input tensor to handle border cases during the convolution operation .
   - This reduces the need for boundary checks and simplifies the access pattern, improving the hardware execution efficiency.
2) Parallelizing the outer axis of Apad and B:
   - [Apad]. parallel (Apad.op.axi[0]) and s[B].parallel(B.op.axis[0]) optimize the schedule by parallelizing the outer loop of both the padded input tensor and the output tensor B.
   - This allows for concurrent execution of multiple iterations, taking advantage of the GPU's massive parallelism to speed up computation.
3) Splitting and vectorizing the inner loop of B:
   - The inner loop of B is split into an outer loop and an inner loop with a factor of 64 (uter , inner = s [B]. split (B.p.axis[0], factor=64)).
   - This creates a hierarchy of loops, where the inner loop is vectorized ([B]. vectorize (inner)), enabling the GPU to perform SIMD (Single In truction, Multiple Data) operations on a group of 64 elements, further improving performance.
4) Binding outer axes of B and Apad to GPU threads:
   - The outer loops of both B and Apad are split into blocks and threads ([B].bind(outer , te . thread_axi (”blockIdx.x”)), [B].bind(inner , te . thread_axi (”threadIdx.x”)), [Apad].bind(outerA, te . thread_axi (”blockIdx.x”)), [Apad].bind(innerA , te . thread_axi (”threadIdx.x”))).
   - This mapping allows the GPU to distribute the workload across its thread hierarchy, improving resource utilization and performance.
5) Pragma directive for unrolling:
   - The pragma directive [B].pragma(outer, ” auto_unroll_max_tep”, 16) unrolls the outer loop of B up to 16 steps.
   - Loop unrolling is an optimization technique that duplicates the loop body multiple times to reduce loop overhead and control-flow instructions.
   - This can help the GPU achieve better instruction-level parallelism and improve hardware execution.

I. Failed Optimizations

The split operation in the intial optimization attempted to split the axis B.op. axis [0] into a number of parts equal to the blockIdx .x thread axis, which is not a fixed integer value, potentially leading to a mismatch between the number of parts and available threads. Instead, it's better to split the axis into a fixed number of parts, as shown in the working code. Additionally, the commented code checks if B.op. axis [0] is not in s[B]. iter_var_attrs , which could be unnecessary or incorrect depending on the context.

J. AutoTVM

For the attempt at implementing AutoTVM to optimize a convolution operation for GPU execution. The function was made by defining a computation graph using TVM's tensor expressions and then creates a schedule object to apply optimizations to the graph. The AutoTVM configuration space is defined by specifying split factors for the loop axes and an unroll knob to control loop unrolling. The function then splits and binds the axes of the computation graph to threads and blocks for GPU execution. Finally, the optimized schedule is returned as well as the placeholders for the input tensors and the output tensor. The @autotvm.template decorator marks the function as an AutoTVM template to be used for search and optimization.

It should be noted that I found AutoTVM to be picky and focused most of my time on optimizing the set functions. In the future it would be helpful and cool to replace the FPGA section with an AutoTVM section as the install is more reliable and it is neat to play around with.

K. Comparison

With the table provided below we can now calculate speedup.

Device	Unoptimized	Optimized	Autotvm
Colab	5.934591e-5	2.0447e-5	–
G2	0.24271	0.13240	–

TABLE II: Summary Statistics for GPU performance

To calculate the speedup factor from unoptimized to optimized, we can use the following formula:

Speedup factor = Unoptimized runtime / Optimized runtime

For Colab:

$$\text{Speedup factor} = \frac{5.934591e-5}{2.0447e-5} \qquad = 2.90$$

For G2:

$$\text{Speedup factor} = \frac{0.24271}{0.13240} \qquad = 1.83$$

Here we note that AutoTVM was not working with an axis error so it was skipped for both colab and G2. Additionally, we also note that the original unoptimized version error-ed out due to a lack of binding and the value placed in the table represents a function with an unoptimized binding.

L. Space-Time Graphs

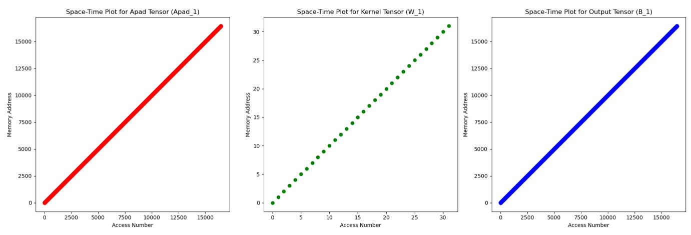

Fig. 2: Space-Time Diagram

GEMM ON GPU

M. Unoptimized Code

The make_gemm_gpu_scheduler function takes three parameters M, K, and N, and computes the matrix multiplication of A and B to produce C, where A is an M x K matrix, B is a K x N matrix, and C is an M x N matrix. This is done using a reduction operation with axis K. The function creates a schedule with block and thread axes for parallelism and caching, and uses shared memory for local storage of A and B tiles. The function returns the schedule object, as well as the placeholders for A, B, and C.

N. Optimizations

We performed the following optimizations for GEMM on GPU:

1) Tiling: The GEMM operation is divided into smaller tiles using the s[C]. tile (x, y, block_size , block_size) line, which splits the x and y axes of matrix C into smaller chunks of size block_size. Tiling allows for better cache locality and reduces memory access latency.
2) Parallelization and binding: The outer loops (yo and xo) are parallelized using s[C]. parallel (yo) and s[C]. parallel (xo), respectively, and then bound to GPU block indices (blockIdx.y and blockIdx.x) using s[C].bind(yo, block_y) and s[C].bind(xo, block_x). This enables different threads to execute different parts of the computation concurrently, improving overall performance.
3) Splitting workloads: The code further splits the workload along the k-axis, dividing it into blocks of size block_size. The inner loops (yi and xi) are also split into subloops (yio and xio) with the same block size. This finer-grained division of work allows for more efficient utilization of GPU threads.
4) Reordering: The loops are reordered using s[C]. reorder (ko, ki, xi, yi, xio, yio). This rearrangement ensures that threads within a block access consecutive elements in memory, improving memory access patterns and cache utilization.
5) Parallelization and binding of subloops: The subloops (xio and yio) are parallelized and bound to GPU thread indices (threadIdx.x and threadIdx.y) using s[C]. parallel (xio), s[C]. parallel (yio), s[C].bind(xio, thread_x), and s[C].bind(yio, thread_y). This enables efficient use of GPU threads and leads to better performance.
6) Local memory caching: Matrices A and B are cached into the GPU's local memory using s.cache_read(A, "local", [C]) and s.cache_read(B, "local", [C]). Caching in local memory reduces global memory access latency, as local memory is faster and closer to the processing units.
7) Compute at: The s[AA].compute_at(s[C], ko) and s[BB].compute_at(s[C], ko) lines specify that the local memory caching of A and B should be performed within the ko loop. This allows the caching operation to be interleaved with the computation, reducing the overall memory access overhead.

O. Failed Optimizations

There were no interesting or failed optimizations that were attempted for this function.

P. *AutoTVM*

The AutoTVM framework used to automatically tune a General Matrix Multiplication (GEMM) operation for a specific target platform. AutoTVM works by first defining a template function for the GEMM operation that takes the dimensions of the matrices and the data type as input, and outputs a TVM schedule for optimizing the operation on a GPU. The AutoTVM task is then created using this template function and target platform. The config space is defined by specifying the axes to be tiled and the maximum number of steps for auto-unrolling. The tuner performs a specified number of tuning trials on this config space, and the best configuration is logged. Finally, the GEMM operation is compiled using the best configuration, and the resulting function is evaluated on randomly generated input matrices. The output of the GEMM operation is compared with the expected output to ensure correctness, and the execution time is measured to evaluate performance. It is included in github for this assignment.

Q. *Comparison*

Subsequently we do the same for GEMM as in previous sections.

Device	Unoptimized	Optimized	Autotvm
Colab	0.088476829	0.006445151	N/A
G2	4.62775	1.97903	N/A

TABLE III: Summary Statistics for GPU performance

To calculate the speedup factor from unoptimized to optimized, we can use the following formula:

Speedup factor = Unoptimized runtime / Optimized runtime

For Colab:

$$\text{Speedup factor} = \frac{0.088476829}{0.006445151} = 13.71$$

For G2:

$$\text{Speedup factor} = \frac{4.62775}{1.97903} = 2.34$$

Note that the AutoTVM function I created has some binding issues and did not compile correctly for me.

R. *Space-Time Graphs*

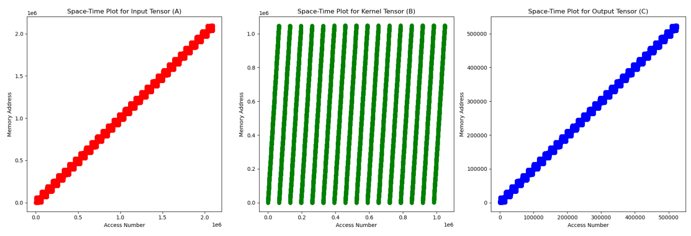

Fig. 3: Space-Time Diagram

2D Depthwise Separable Convolution on GPU

S. *Unoptimized Code*

The function make_dwsp_conv2d_gpu_scheduler implements a depthwise separable 2D convolution operation for the GPU, taking as input the number of batches (B), input channels (C), input height (H), input width (W), and kernel size (K). It defines placeholders for the input tensor inp and the kernel tensor ker, and two reduction axes, rkh and rkw, for the kernel height and width dimensions. The input tensor is padded with zeros to preserve the spatial dimensions of the output tensor, and the output tensor out is computed by iterating through the height and width dimensions, performing a depthwise convolution using the padded input tensor inp_pad and the kernel tensor ker, with a summation over the kernel height and width dimensions. The function creates a schedule s for the depthwise separable 2D convolution operation, defining parallelism at different levels, such as splitting the height and width dimensions of the output tensor into multiple parts, and binding these parts to the appropriate thread axis (blockIdx or threadIdx) to parallelize the computation and exploit the GPU architecture for faster execution.

T. *Optimizations*

We performed the following optimizations on the 2D Depthwise Separable Convolution on GPU:

1) Padding: A padding operation is performed on the input tensor using the te.compute function. The input tensor is zero-padded by pkh and pkw pixels on each side to ensure that the convolution operation is performed properly at the borders of the input tensor.
vbnet Copy code
2) Convolution: A convolution operation is performed on the padded input tensor using the kernel tensor. The output tensor is computed using the te.compute function, and the te.sum function is used to perform the convolution operation.
3) Parallelization: Parallelization is performed on the output tensor to optimize the computation for GPUs. The te.create_schedule function is used to create a schedule for the output tensor, and the s[out].parallel function is used to parallelize the computation across multiple threads. The te.thread_axis function is used to specify the type of parallelization to be performed.
4) Loop splitting: The s[out].split function is used to split the loop over the second axis of the output tensor into two loops, which are then parallelized separately. This is done to optimize the computation for GPUs.

Overall, the optimizations that are performed in this function aim to improve the performance of the convolution operation on GPUs by reducing memory access and increasing parallelization. The padding operation ensures that the convolution operation is performed properly at the borders of the input tensor, while the parallelization and loop splitting operations optimize the computation for GPUs by distributing the workload across multiple threads.

U. *Failed Optimizations*

Like GEMM there were no failed optimizations, but in discussions with others in the class it was noted that the scheduling of this function is not yet optimal and could be improved.

V. *AutoTVM*

AutoTVM was not included given the complexity and time constraints of the assignemnt.

W. *Comparison*

We now calculate the speedup for the 2D Depthwise Separable Convolution on GPU.

Device	Unoptimized	Optimized
Colab	0.0015987	0.000170368
G2	4.461677s	3.09891

TABLE IV: Summary Statistics for GPU performance

To calculate the speedup factor from unoptimized to optimized, we can use the following formula:

Speedup factor = Unoptimized runtime / Optimized runtime

For Colab:

$$\text{Speedup factor} = \frac{0.0159}{0.000170368} = 93.18$$

For G2:

$$\text{Speedup factor} = \frac{4.461677}{3.09891} = 1.44$$

X. *Space-Time Graphs*

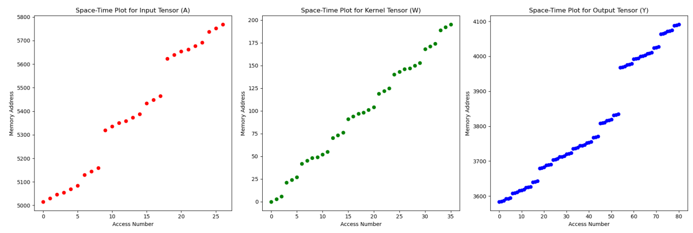

Fig. 4: Space-Time Diagram

Further Directions

In conclusion, TVM provides a powerful tool for optimizing DNN primitives on different devices. The optimizations performed on the primitives had a significant impact on the runtime and speedup of the computation. The use of TVM IR and space-time diagrams can aid in the choice of tiling and vectorization optimizations, however there is a lot to learn/improve upon in this regard. It would interesting to further explore the generation and refinement of the space-time graphs in addition to experimenting with more optimizations and various values of M and N as I have seen others done in this assignemnt.

Disclaimer

All code and other resources used for this assignment can be found on the A3 repository and in the TVM documentation. I used various LLM models (ChatGPT, GitHub Copilot,...) to help with function debugging and assumption making. I also talked with various members of the class to brainstorm ideas and help create interesting optimizations.