C Split Globally: High Performance Convolution Kernels for CNNs

Erik Barton
University of Utah

UUCS-22-001

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

5 January 2022

Abstract

Convolutional Neural Networks are a common deep learning architecture for image processing reliant on internal convolution operations. To improve overall performance, these convolution operations must be highly performant. GPUs provide a hardware architecture that accommodates substantial levels of parallelism that can improve operational performance. However, code must be carefully developed to obtain this potential. This thesis develops high performance GPU kernels for this purpose by using careful analysis of the convolution problem and its implementation on the GPU. By parameterizing the execution space, this thesis shows that the kernels can be made to adapt to specific problems in architectures. Finally, it is shown that the developed kernels are, in many common cases, more performant than comparable methods while still having great potential for further refinement.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

CONVOLUTIONAL NEURAL NETWORKS AND THEIR GPU KERNELS

Image processing is an important area of computational research. One of the leading algorithms in image processing is the convolutional neural network (CNN). CNNs are an effective form of image processing and identification built around convolution operations. Executing this operation as efficiently as possible is of great importance to quickly training such neural networks and to their subsequent use in image prediction. As such, the goal of this research was to identify ways to improve convolution performance specifically on GPU architectures by creating a new set of kernels whose execution could be adjusted by various predetermined parameters.

1.1 Convolutional Neural Networks

As with most neural network systems, CNNs apply layers of weighted filters to develop data into a state where it is ready for evaluation [13]. In CNNs this is mostly done through the use of convolution operations. The operation is built around taking a matrix of some size less than or equal to that of the input image matrix and convolving it over all possible indices within the input [13]. This smaller matrix is often termed the filter, kernel, or weights [1]. At each index, the values of the filter are multiplied with the corresponding values in the input matrix and summed. This final value is then put into a single element in the output matrix. The filter is then moved to a new location jumping a distance termed the stride. The convolving process continues until all possible indices where the filter can be centered in the image have been evaluated. To allow for easier indexing, the input is often padded with additional zero values along the outer and lower edges to allow for the filter to easily fit in the rows and columns.

In the case of most images, there are multiple layers in the image often called channels [1]. These channels are often separated into the various color values for RGB. In the

convolution process, all of these channels are applied to corresponding filter channels and placed in the same location in the output. This essentially compresses all of the channel values into a single result as dictated by the filter. Also, in most instances of CNN convolution, there are several filters rather than only one applied to the input. This allows for different attributes of the image to be identified by each filter [1]. Therefore, the whole operation ends up performing work on tensors of three and four dimensions for a single image. These tensors can be viewed in figure 1.1.

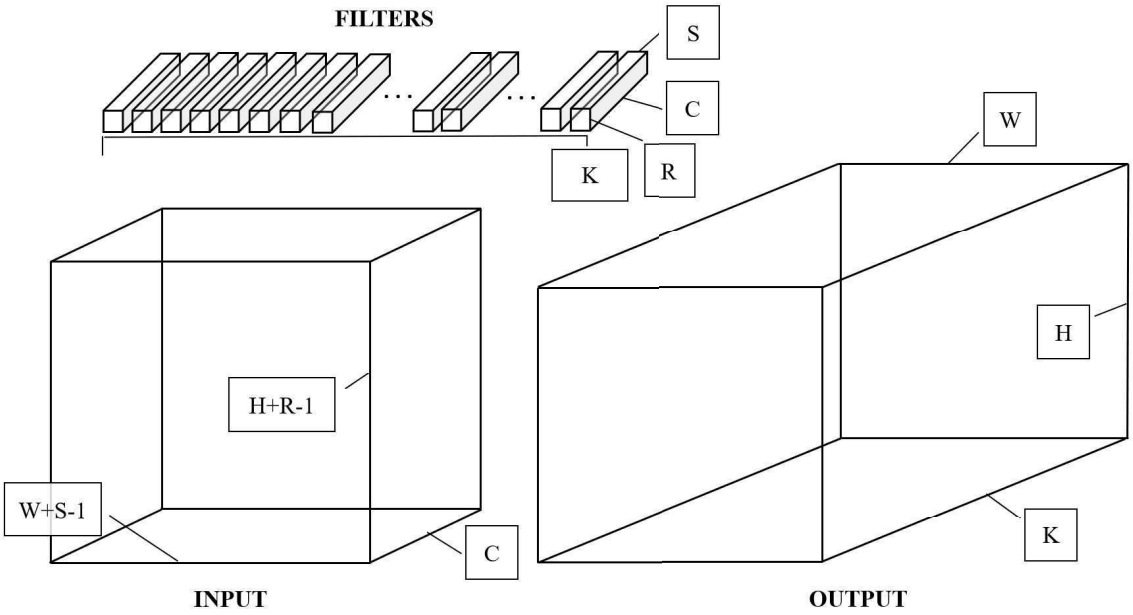

Figure 1.1. CNN tensors for batch size of one.

This whole convolution process is then repeated through several layers in the neural network called the pipeline. Successive applications of layers with distinct filters allows for later filters to evaluate larger regions of the image. Therefore, these later filters are trained to identify larger features in the image by building off of the smaller features identified by previous filters [1]. This convolution process is at the heart of CNNs (hence the name) and is what gives them their effectiveness.

On a more applicable level, this entire convolution process can be coded as a simple multiplication and addition operation surrounded by successively nested loops. An ex-

ample of this code can be seen in the figure 1.2. Due to the structure of the operation, these loops can be permuted into any order and tiled in any way [9]. This means that although the basic structure is simple, the possible options for increasing performance are enormous. Due to its central nature and repeated application, performance is an important consideration.

```
for(n = 0; n < Nn; n++)
 for(k = 0; k < Nk; k++)
  for(c = 0; c < Nc; c++)
   for(r = 0; r < Nr; r++)
    for(s = 0; s < Ns; s++)
     for(h = 0; h < Nh; h++)
      for(w = 0; w < Nw; w++)
       Out[n][k][h][w] +=
        In[n][c][h+r][w+s]*Ker[k][c][r][s]
```

Figure 1.2. CNN loops including a batch index (n) for multiple images. [9]

1.2 GPU Architecture

GPUs are a form of arithmetic hardware designed for high levels of parallelism and run functions called kernels [10]. Numerous parallel cores allow for execution of single instruction multiple thread operations in what is termed a warp [10]. Warps of separate threads can, in this way, simultaneously execute operations providing concurrency. Although these systems are designed with warps of 32 threads in mind, not all threads must execute the same instruction on modern architectures. However, this loses potential concurrency.

Each thread itself is assigned numerous registers at launch which are retained by the thread until completion [10]. This retention of registers allows different warps of threads to be switch in and out of a running context at zero cycle speeds. Because of this, many warps can be run together, filling available resources, and switching out with one another during high latency instructions such as memory fetches. These warps are organized into groups of threads called thread blocks. Numerous thread blocks can be used in grids to help better define the operation space. The thread blocks are also assigned to different streaming

multiprocessors (SMs) that contain the cores, registers, and other memory. These separate SMs work independently further increasing parallelism.

Further performance enhancing techniques are common in GPUs such as instruction pipelines to help hide instruction latencies and special memory units called shared memory which allow for manually controlled cache with high warp sized throughput [10]. All of these different hardware features make GPUs ideal for executing highly performant and parallel operations. However, in order to obtain such performance, code must be carefully designed for the GPU. Also, various architectural differences found in successive generations of GPUs make some considerations vary on a per architecture basis.

CHAPTER 2

RELATED WORK ON CNNS

Due to the demand for image processing, various attempts to improve the performance of CNN kernels have been attempted. These approaches vary for different hardware options. However, many are built on modifying the previously mentioned nested loops into a new format designed to get the most out of the specified hardware.

2.1 CPU Approaches

On CPU hardware, the hardware resources are organized into simple cache levels with a relatively small number of CPU cores. With this structure, and almost all others, data movement is by far the most time expensive operation. As such, the approaches to CNN kernels generally center around most efficiently moving data through the different levels of the cache hierarchy.

Many solutions to CNN kernel optimization on CPUs utilize a system called auto-tuning. In this process, a search space of various combinations for options of tile size and loop order are created. Then the system proceeds to search this created space by modeling or actually testing one combination at a time [2]. Through this process, more performant combinations can be identified and those options can be used in larger data sets or different input values.

This process can be viewed in a similar way to most standard optimization processes such as the well known Gradient Descent. By testing various options, performance can be evaluated in an unfamiliar domain. More "intelligent" choices can be made between each test to reduce the number of options tested. However, regardless of intelligence, the generated space must be searched to some extent to find the most optimal choice and any gaps in that generated search space are ignored.

Therefore, in a large design space, such as provided by the convolution problem, the full option space can exceed hundreds of trillions of choices [9]. This means that any

reasonably timely auto-tuning software will be forced to ignore many of the options. However, newer techniques can use analytical modeling to search the entire design space to optimize for data movement.

By using models of the data movement costs for various tile structures and loop orders, options can be quickly evaluated. Evaluating different levels of tiling and permutations from the first level up, also allows for early pruning of the design space. Putting this together for all loop levels allows the entire design space to be evaluated or pruned from the search space without requiring actual test runs [9]. As such, this technique can quickly find more optimal code orderings for the CNN problem.

2.2 GPU Approaches

In the case of the more complicated GPU architectures, the number of options for code structure increase even further. However, this structure also provides the potential for further increases in performance due to the GPU's heavily parallelized nature. Therefore, developing optimized GPU kernels is a key component of making fast CNN kernels.

Many current solutions to GPU CNN calculations rely on mapping the problem down to a two-dimensional matrix multiply problem [7]. This process is frequently termed GEMM. By carefully mapping the input and filter values to two dimensional tensors, standard matrix multiplication can be performed and still yield the same results as direct convolution. One example of this method is the open-source CUTLASS CNN solver [5]. However, this approach is also one of the options in the cuDNN package, which is a leading set of closed source CNN software [7].

The mapping of the input tensors is necessary to allow for efficient evaluation and data movement on GPUs whose structure is designed to be most efficient in linear access patterns and in the evaluation of linear operations. Use of tensor cores can allow for some level of two-dimensional evaluation and, as such, they are frequently utilized for these problems. However, the 1D and 2D focus of hardware computational components essentially forces this dimensionality reduction in solutions.

In order to obtain faster access to memory, shared memory is often employed. However, the structure of shared GPU memory requires careful saving and accessing patterns to maintain performance by avoiding access conflicts. In the case of CUTLASS, this is

done by remapping the loaded memory so that any adjacent thread accesses will map to non-conflicting locations [5]. This is done through the use of careful access patterns and an XOR operation to remap loaded values.

Even with these and other carefully constructed components, GPU CNN kernels still suffer from reduced performance when being utilized on arbitrary architectures or if not carefully tuned to the problem's needs by the implementer [7]. The enormous design space and varying hardware constraints mean that options continue to exist to developing more performant and generalizable kernels.

2.3 Alternate Approaches

Additional approaches to quickly solving the convolution operation also exit. Many of these make use of a change in problem space to leverage mathematical advantages found in the frequency domain. In the frequency domain, the convolution operation can be reduced to simple summed multiplication operations [7] [8]. This vastly reduces the total number of operations required to complete the convolution operation, but also means a kernel must perform transformation operations to move into and out of the frequency domain. These approaches often utilize the fast Fourier transform or the Winograd algorithm.

The downside to this approach is the aforementioned requirement to move in and out of the frequency domain. This can cause loss of performance due to the additional operations required, especially for small problem sizes. As such, various options have been and are being explored to avoid this domain movement. One such approach is to simply keep resulting values from the frequency domain convolution operation in the frequency domain [11]. By doing so, later steps in the CNN problem must also take place in the frequency domain, but operations are saved in the convolution step.

cuDNN provides implementations for both fast Fourier transform based algorithms and Winograd based algorithms along with their GEMM implementation. However, it can be seen that by varying problem dimensions, performance resulting from all of these methods will vary widely [7]. For this reason, this research explored a more model driven approach.

CHAPTER 3

KERNEL DEVELOPMENT

To find and develop more optimal CNN GPU kernels this research focused on developing modeling that would reflect the expected execution of the kernel on the given GPU. Through this modeling, parameters would be chosen that would adjust kernels to obtain more optimal code behavior. The goal was to make these kernels in such a way that the modeling of the problem would allow for adaptation of the kernel for different problems and architectures and thereby produce an increase in performance.

3.1 Assumptions and Design Constraints

Due to the exceedingly large design space, several assumptions were necessary to simplify kernel development. Care was taken while specifying these assumptions to only narrow the design space to a limited extent and still allow the developed kernel room to adapt to the most common use cases in CNN's. However, these assumptions did remove some of the completely arbitrary functionality and adaptability that was desired for the program. This was deemed acceptable for the initial exploration into this kernel design.

The first assumption made was that the K dimension of any given problem would always be some multiple of 32. This assumption was true for all but one of the layers of the CNN pipelines tested and is common in most such pipelines [6], [12]. The benefit of this assumption was that it provided assurance that a full warp or set of full warps could always correspond to the K dimension of the design space.

The second assumption was that any given input matrix would be padded to the correct value for the expected output size given the filter size. For example, if output W was expected to be 2 with a filter S of 3 for a stride one operation, then the input width would have to be 4. This assumption removed the need for a given kernel to check for boundaries during computation or apply padding prior to computation. Along with this assumption, it was also assumed that image batch size would only be 1. This meant that

the resulting kernel would be developed considering only the most limited input size and any need for a batch system could be implemented as part of a wrapper call to the kernel.

The third assumption was that an automated setup time for a given input size could be any amount of time within a limited number of hours reasonable for the problem size. Within that time, an appropriately designed kernel should have been selected from the design space and compiled. It was also assumed that multiple kernel designs could be used in this selection process as long as the final resulting compiled code would function for the given input. This assumption freed up time for input evaluation and computation. It also removed the need for certain kernel implemented checks for parameters like stride length. The assumption would not prevent the code from being used in the most common case of CNN convolution which uses predetermined input layer sizes defined prior to layer training or utilization.

Fourth, was that problems would be orientated in memory according to a specified parameter order and that results would have to be return in a similar order. In this case, the input tensor would be required to have an order of: C, H, W. The output tensor would have an order of: K, H, W. Lastly, the filter set tensor would have a C, R, S, K ordering. These orderings would of course dictate what dimensions were most closely grouped in linear memory. This in turn would affect what memory access patterns would be more or less efficient.

Finally, any kernel would, of course, be constrained by the resources available on the desired runtime GPU. Since different GPU architectures exist, available resources vary. However, any given compiled kernel would be expected to function within the available resources on the specified GPU where the resource specifications were known prior to kernel selection and compilation.

CHAPTER 4

REGISTER LEVEL MODELING AND IMPLEMENTATION

Since registers are the fastest and most highly utilized portions of the GPU, kernel design began at this level. The goal was to develop a register utilization pattern that would get the most reuse out of any one loaded value while still accounting for hardware constraints. Register speed makes reuse at this level beneficial for performance since access is accomplished at nearly instantaneous speeds. Since registers are allocated on a per thread basis, this meant that any sort of register reuse would require each thread to compute multiple values also known as a tile.

4.1 Theoretical Modeling for Register Tiles

Since output elements in the convolution operation are created by successive operations on different input and filter elements, output elements are the only elements that are consistently reused throughout the computation of any one final value. Thus, keeping a stationary set of output elements in registers allows for element access to be unconstrained by higher memory operations. Taking this approach, there are only three dimensions any thread can be responsible for in output: K, H, and W. Tiling of this nature can be seen in figure 4.1.

However, these output elements are also dependent on various elements from input and the filter set. These elements might also obtain reuse in the computation. Therefore, allocating some of these to registers is beneficial to performance as well. Since the output elements determine what elements are required, the necessary input and filter elements are easily identified. The tile shapes resulting in input and the filter tensors can be seen in figure 4.2.

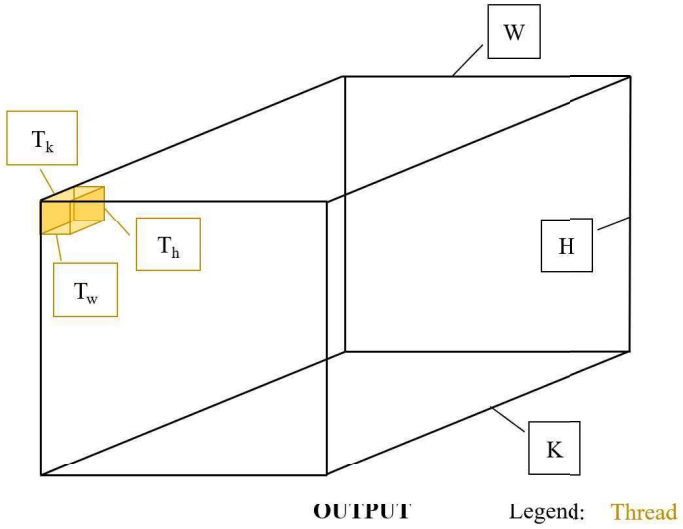

Figure 4.1. Sample tile space for a single thread in the output tensor.

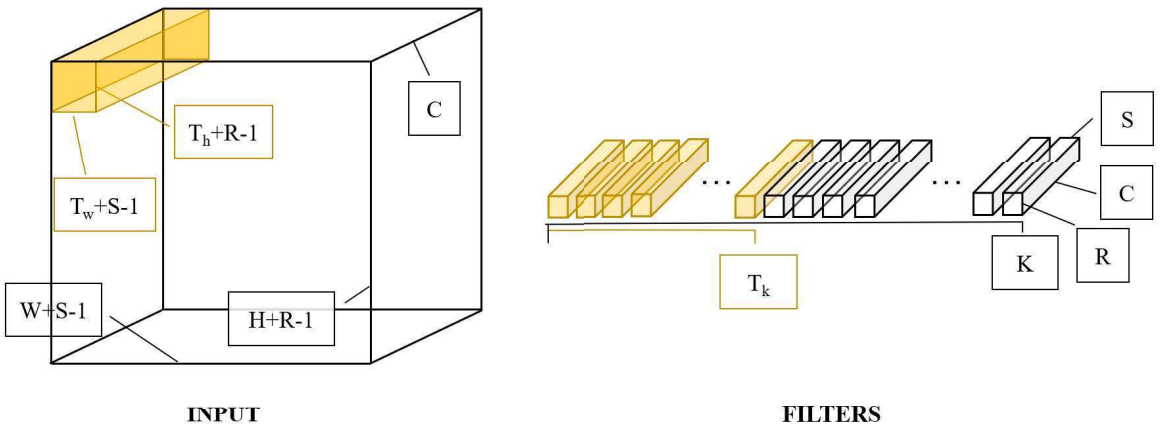

Figure 4.2. Sample tile space for a single thread in the input and filter tensors.

4.2 Execution Pattern

Although tiling in all tensors can be beneficial, it requires many registers. Furthermore, in the operations performed in convolution, tiling in both the input and the filter tensors

can actually result in wasteful use of registers. Since math operations occur on individual registers, and assuming loads are performed for each register individually, retention of both sets of input in registers does not provide reuse.

This idea is more clearly viewed in the simplified matrix multiply case. In matrix multiply, the same multiply-add operation is performed as in convolution. As can be seen in figure 4.3, the output plane depends on a row and a column from the inputs. However, since values are computed sequentially, only loading one set of inputs fully will obtain the same reuse as loading both sets. The same is true in the convolution problem since the core operation is the same.

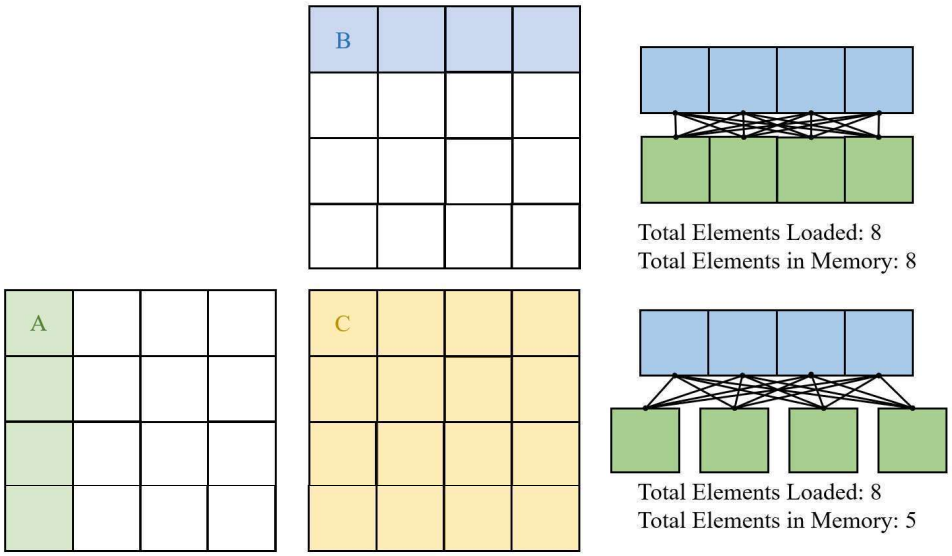

Figure 4.3. Matrix matrix multiply example of tile reuse where a column in A and a row in B partially contribute to all values in output.

Thus, by loading only either input fully or the filter set fully, register pressure can be decreased in this tiling scheme. However, not both options are viable. As seen in figure 4.4, the reuse scheme for a single input element is substantially more complex than the scheme for a single kernel element. Since the goal of the program is to spend as much of its time as possible on computation rather than indexing, the single kernel load is the best approach. This simplifies the indexing scheme to the one seen in figure 4.5.

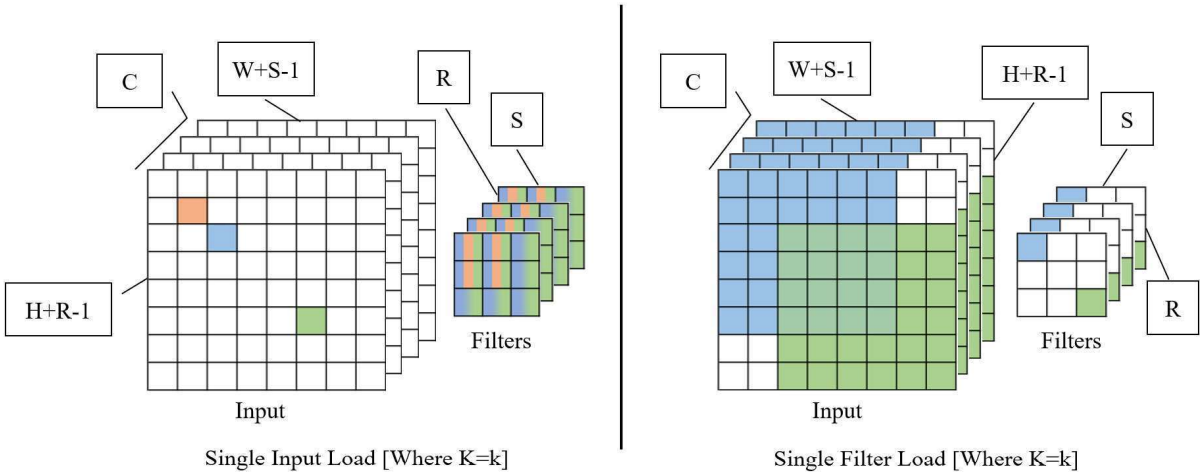

Figure 4.4. Sample of the reuse pattern of loaded input elements in the filter set (left) and a sample of the input reuse of loaded filter elements in input (right).

```
for c in range(C):
  for k in range(K):
    for r in range(R):
      for s in range(S):
        for h in range(r, H+r):
          for w in range(s, W+s):
            Out[h-r][w-s] += In[h][w] * Filters[c][r][s]
```

Figure 4.5. Sample of the logical CNN loop structure with single filter element traversing the input.

4.3 Calculations

With these selections, it is finally possible to begin reasoning about the performance implications of various parameter selections. The most useful estimation of performance at this level is the operational intensity achieved by any one thread. The operational intensity is the ratio of the number of operations that can be executed with respect to the number of elements loaded from memory and can be calculated based on the parametrized tile dimensions. The resulting equations can be seen in figure 4.6.

Using operational intensity, various tile sizes used in the division of the overall problem can be quickly compared to find the most optimal. To simplify the number of possible

$$OI = ((r*s)*T_k*T_w*T_h*T_c) / ((T_h + r - 1) * (T_w + s - 1) * T_c + r * s * T_k * T_c)$$

Figure 4.6. Thread level operational intensity equation based on tile size parameters.

dimensions, tile sizes are limited to sizes that perfectly fit the output. This limits the option space, but also removes the need for partial tile considerations and implementations.

4.4 Register Level Implementation

Implementing the register level tiling structure required only slight modification from the original set of nested loops. For each c value, an input plane of values is loaded into a register array. For each k value, the input plane is used with loaded filter values for the computation of an H and W output plane. One filter value is loaded for each (R, S) before computing the output plane using the current input plane. This limits the filter set to only using one register at any given time. Output is computed in planes, but is saved into a three-dimensional vector of registers with dimensions T_k, T_h, and T_w.

A simplified version of this implementation can be seen in figure 4.7. It can be seen that each input plane will be used for all output planes in the T_k dimension. The kernel elements will obtain reuse for all elements in a given output plane with dimensions T_h and T_w. Output elements will be reused for all C, R, S, T_k, T_h, and T_w operations. All these behaviors match those desired from the theoretical modeling.

```
for (c = 0; c < C; c++) {
    for (k = 0; k < T_k; k++) {
        for (r = 0; r < R; r++) {
            for (s = 0; s < S; s++) {
                oneKern = Kernel[ (kIndex + k)*C*R*S + (c)*R*S + (r)*S + (s) ];
                for (h = r; h < T_h + r; h++) {
                    for (w = s; w < T_w + s; w++) {
                        regOut[k][h-r][w-s] += regIn[h][w] * oneKern;
                    }
                }
            }
        }
    } // End k
} // End c
```

Figure 4.7. Loops for thread level tiling with singly loaded kernel elements.

Although all of these accesses seem to be clear, due to the nature of register allocation in GPUs, this simple array access structure posed some problems in many tile sizes. If the compiler was unable or unwilling to fully unroll any of the loops in K or below, then the compiler could also not resolve the generated arrays to registers. Instead, it was found that the compiler would move these arrays into local memory, which is essentially another name for global memory. This caused any performance improvement to be lost due to slow local memory access times. Resolving this problem required manually adding compiler directives to these loops specifying the complete unroll factor of each loop. With this specification in place, performance improvement was achieved over the base kernel.

CHAPTER 5

THREAD BLOCK LEVEL MODELING AND IMPLEMENTATION

Continuing from the performance gains produced from register tiling, further opportunities for performance gains remained. Since threads are executed as parts of warps and warps as parts of thread blocks, levels of the GPU structure were still available to implement code beneficial for overall performance. The main benefit achieved at these larger data divisions would be through the use of shared memory.

5.1 Theoretical Modeling

Previously, only the work done by a single thread was considered for reuse and was firmly limited by the registers available to that thread. At the thread block level, shared memory can be utilized to further increase reuse of memory loads while not increasing register pressure. The various tensors tiled previously can again be tiled using parametrized values relative to the thread block size.

In figure 5.1 the simple extension from thread tiling to thread block tiling can be seen. Since it was assumed that the K dimension would always be a multiple of 32, it was safe to allocate warps in this dimension. Tiles in H and W can then simply be extended from their previous version to encompass all the work done by adjacent warps in those directions.

By grouping these threads together in this way, substantially more input reuse can be obtained. Since shared memory is shared among all threads in a block and since all output elements are dependent on corresponding input elements from every c value, any input element loaded to shared memory can be reused by all threads in the k dimension with corresponding H and W values. This means any input element loaded can obtain reuse for T_k times B_k output elements (where B_k is a multiple of 32). Thus, reuse in the K dimension can be improved by B_k times over simple register tiling.

As with the register level tiling, calculations can be made for performance indicative

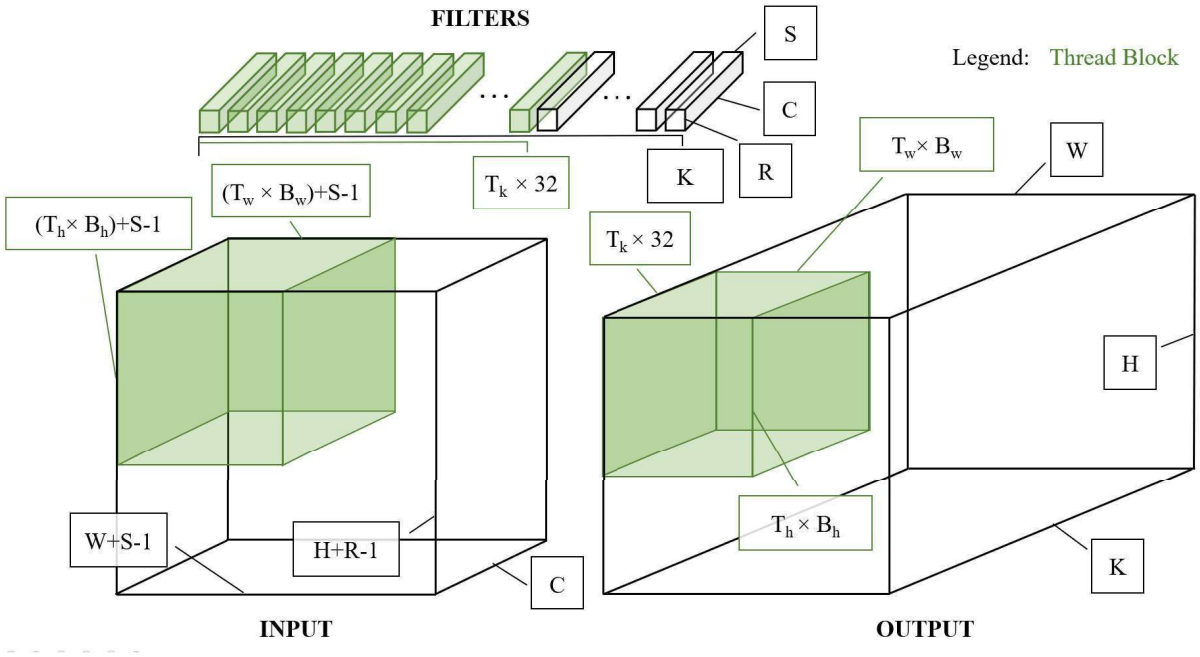

Figure 5.1. Sample tile space for the entire thread block in all three tensors where $B_k = 32$.

values like operational intensity. As seen in figure 5.2 these equations are very similar to those found in figure 4.6. However, they incorporate the additional parameters relevant to a thread block's behavior. When calculating these values for various configurations, the resulting numerical values are substantially higher than the values for register level operational intensity. However, since shared memory has higher latency than registers, the values of operational intensity for this level cannot be compared with those of the register level.

$$\text{oppsSM} = B_w * B_h * T_w * T_h * W_k * T_k * r * s * W_c$$
$$\text{sharedVolume\_In} = (B_w * T_w + s - 1) * (B_h * T_h + r - 1) * W_c * T_c$$
$$\text{OISharedMemory} = \text{oppsSM} / \text{sharedVolumeTotal}$$

Figure 5.2. Thread block level operational intensity equations.

5.2 Thread Block Level Implementation with Shared Memory

Since thread blocks are divided into up to three dimensions (X, Y, and Z), the dimensions of the thread block itself can match the dimensions of the output tensor. However,

warps in a thread block are allocated from contiguous regions of the linearized block dimensions. This means that since x is the fastest varying index, sequential threads in x are grouped into a warp first.

Therefore, to perform coalesced access at any level, access has to be performed with this linearized format of the thread block in mind. By making the thread block 32 in the x dimension, the y and z dimensions can be easily used to identify unique coalesced warps of threads in the thread block. This 32-thread dimension also easily maps to the multiple of 32 found in the k dimension making a mapping of X to K an obvious choice. Thus, for the purposes of computing output, X is mapped to K, Y is mapped to W, and Z is mapped to H. As seen in figure 5.3.

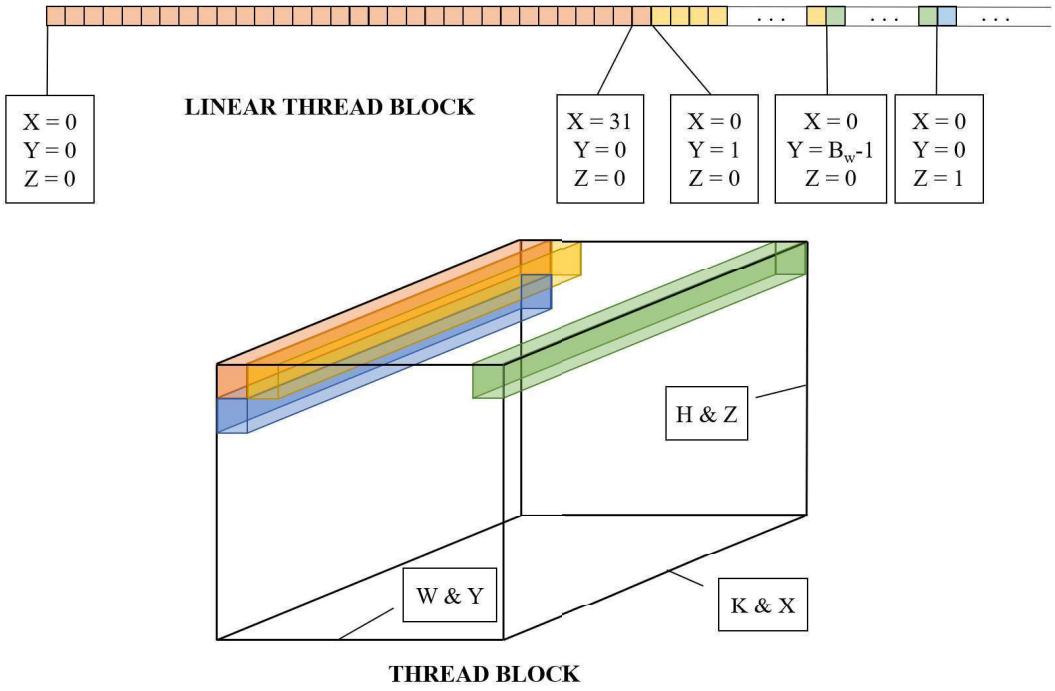

Figure 5.3. Thread block mappings in linearized and non-linearized formats.

5.3 Input Load Structure

Even with the selected X to K thread block mapping structure, there was no set requirement that the mapping be used in all operations. This freedom was useful when

working with the input and output tensors. These tensors were assumed to have a the fastest varying index in the *W* dimension. Thus, coalesced access would require warps to work in that dimension rather than in the *K* dimension.

To implement the shared memory load for the input tensor, coalesced access is important in both the load from memory and the save into shared memory. It is also important to make any indexing in either structure simple so that performance is not bogged down by computational overhead. Therefore, a simple approach was selected for importing input.

As seen in the register level loop structure (Figure 4.7), with no tiling in the *C* dimension, one *H*-*W* plane from input can be loaded for each *c* and used for all subsequent operations in the nested loops. The process selected for loading a given plane needed by a thread block is seen in figure 5.4. By orientating the warp in the *W* dimension for this operation, coalesced access is assured in global memory, and bank conflicts in shared memory are prevented.

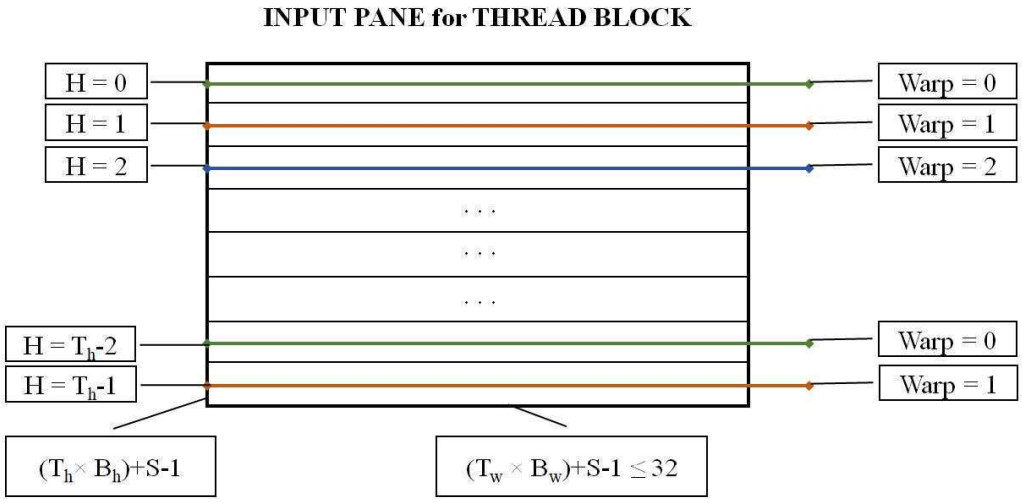

Figure 5.4. Load pattern for an input plane from global to shared memory.

However, in many cases, the *W* dimension will not be 32 or one of its multiples. In fact, it is frequently less than this. In this instance, several threads in the warp will not have items to load from global memory. This is wasteful from a thread perspective, but the ease of indexing produced by using this approach makes this cost more than acceptable.

Conversely, in the cases where the W dimension needed by a thread block is larger than 32, this process begins to experience more substantial inefficiencies. To account for this larger number (but continue to obtain coalesced access), more complicated indexing must be implemented, or warps must perform a second load where a majority of the threads are likely to be inactive. Both of these are degrative to performance, so in this implementation the maximum block size loaded to memory was limited to 32 in the W dimension. This reduced the possible tile size space, but allowed for simple, performant indexing to be used.

The input elements now found in shared memory still needed to be loaded to registers. Using the selected X to K view of thread blocks for this operation proved very useful. As can be seen in figure 5.5, all threads in a given warp can load the same input elements from shared memory since that element is to be reused in the K dimension which also maps to the warp. Since all threads access the same element, there are no bank conflicts in shared memory. Reuse is also obtained in the tile overlaps between different threads. The amount of reuse is less substantial than in the K dimension, but is still non-zero.

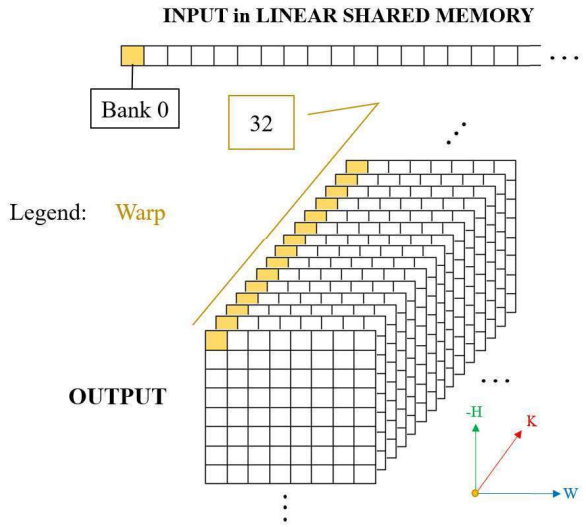

Figure 5.5. Mapping of elements found in shared input memory to affected elements in output global memory for a warp.

The most important part of this access pattern is the difference between the number

of global memory loads executed by each thread verses the register only pattern. In the best case, each warp in this pattern need only perform one load from global memory. In the register only version, each warp must perform $(T_w + S\text{–}1) * (T_h + R\text{–}1)$ loads for a given input plane. This substantial difference is offset by the fact that there are still $(T_w + S\text{–}1) * (T_h + R\text{–}1)$ loads per thread from shared memory in the thread block version, but those loads are much faster than global memory netting an overall performance gain.

5.4 Tile Pattern in K

Actual computation of the problem could be completed without substantial change in the existing register code. However, the orientation of warps in the K dimension meant that loads from global memory for the filter sets would also be executed along that dimension. Having assumed that this dimension was innermost, there was the potential for coalesced access during this load.

Unfortunately, as can be seen in figure 5.6, the output execution structure developed in the theoretical model would not allow for this coalesced access. With each thread in a warp performing some set of K planes prior to the start of the next thread's work, access would always be dispersed by T_k between threads. To solve this problem, the execution structure was changed from warps performing 32 sets of T_k sized tiles to performing T_k sets of 32 wide tiles. Figure 5.7 shows how this allows for coalesced access into the filter memory.

However, this new execution structure did require some changes to the code as seen in figure 5.8. These changes are very minor but also cause the resulting register values in a thread to take on a new meaning. Figure 5.9 displays how planes in the register three-dimensional view now correspond to output planes separated by 31 other planes in the output. This jumping causes the register logical structure to become uncoalesced, but since registers in hardware are not accessed by other threads, this uncoalesced nature has no impact on performance.

5.5 Output Save and Resolving Bank Conflicts

Following computation, saving computed values back to the output tensor in global memory could also receive benefits from the use of shared memory. By placing all of a

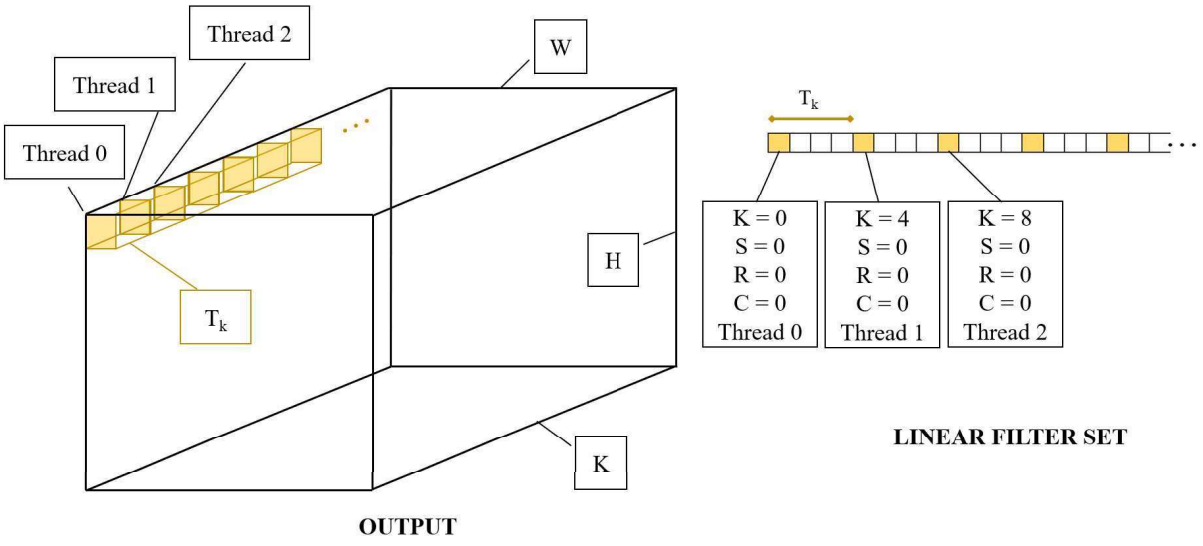

Figure 5.6. Depiction of thread tiles within the same warp in global output memory with coalesced tiles.

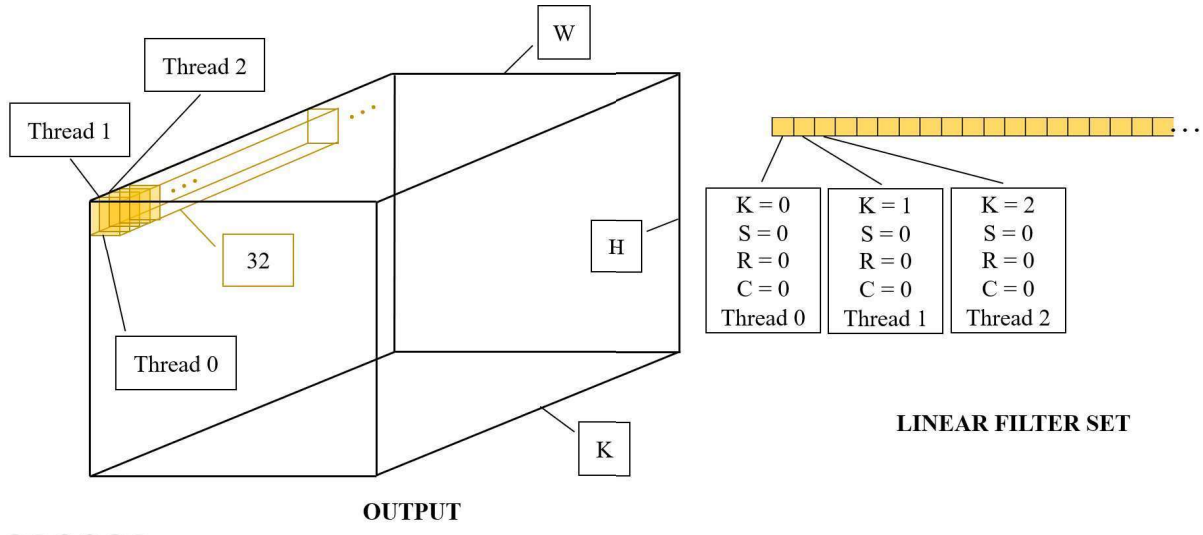

Figure 5.7. Depiction of thread tiles within the same warp in global output memory with uncoalesced tiles.

thread block's register values into shared memory, values in the *K* planes could be saved back to global memory in a nearly identical way to the input load. A diagram of this process was seen previously in figure 5.4. This approach once again allowed for coalesced

```
for (c = 0; c < C; c++) {
    kReg = 0;
    for (kGlobal = 0; kGlobal < kWarpReach; kGlobal+=32) {
      for (r = 0; r < R; r++) {
        for (s = 0; s < S; s++) {
          oneKern = Kernel[ (c)*R*S*K + (r)*S*K + (s)*K + (kIndex + kGlobal) ];
          for (h = r; h < T_h + r; h++) {
            for (w = s; w < T_w + s; w++) {
              regOut[kReg][h-r][w-s] += regIn[h][w] * oneKern;
            }
          }
        }
      }
      kReg++;
    } // End k
  } // End c
```

Figure 5.8. Thread block level code implementation with uncoalesced register tiles.

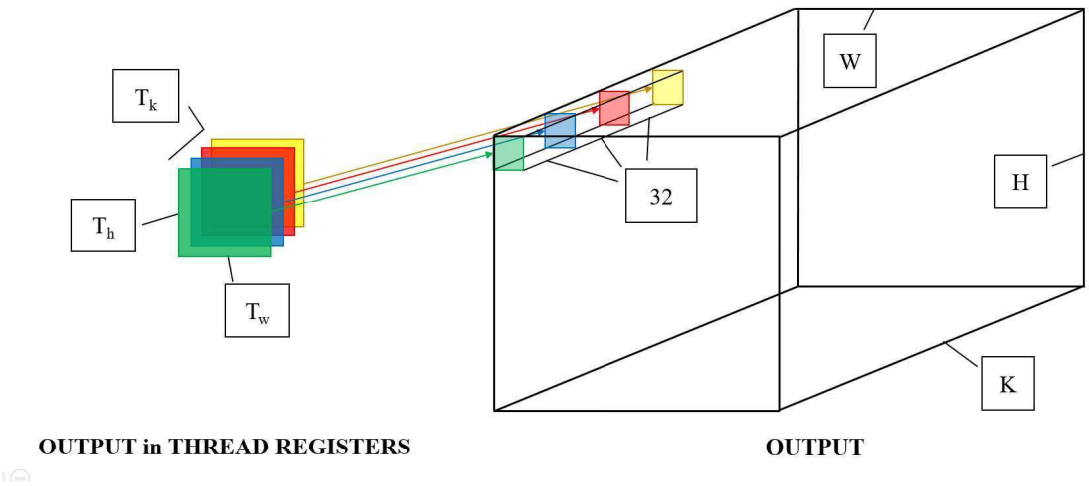

Figure 5.9. Mapping on thread level output planes to global output.

memory access with easily computed indices and resulted in shared work among the threads.

However, this save back operation differs from the load operation is several important ways. Most obviously, while the input load was done as individual planes nested inside of the C loop, this save must be conducted on a three-dimensional tensor of planes in the K dimension. The total data volume for all of the threads in a thread block is $T_k * T_h * T_w *$

$32 * B_h * B_w$ which is too large to efficiently hold in shared memory. Loading the entire block would also require each thread block to make extensive use of the shared memory pipe making concurrent use with other blocks difficult.

Therefore, like in the input load, a better approach was to break the output save into separate stages that were nested inside of a loop. In this case, though, the loop would be traversing the K dimension instead of C. Rather than performing this operation in steps of size one, grouping this operation into steps of size 32 would make total shared memory volume smaller by a factor of T_k but also allow for better warp wide operations. Figure 5.10 shows how the total thread block output space is subdivided by the corresponding loop structure into these smaller groups.

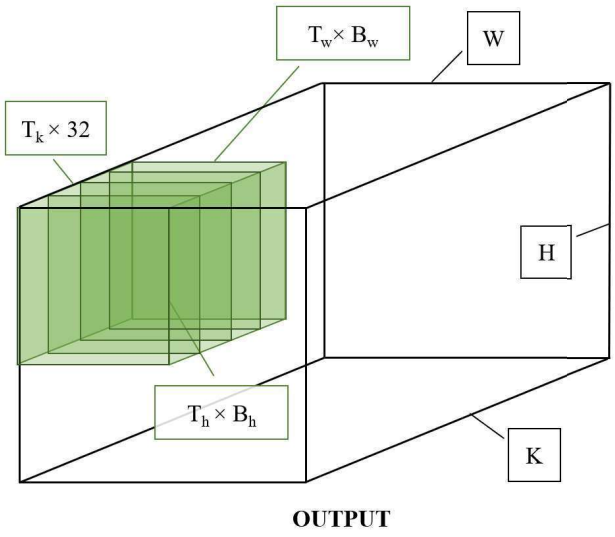

Figure 5.10. Thread block subdivisions for save back operation in global output where $T_k = 4$.

Initially, the loop pattern for the save had the same order and shared memory structure as the load. Unfortunately, this caused bank conflicts on the save operation from registers to shared memory. Since W is the innermost shared memory dimension it will be divided into the shared memory banks first. When $T_w * B_w$ is not one, then some or all banks will be used for values in the next $T_h * B_h$ row. This will cause bank conflicts as is clearly seen in figure 5.11.

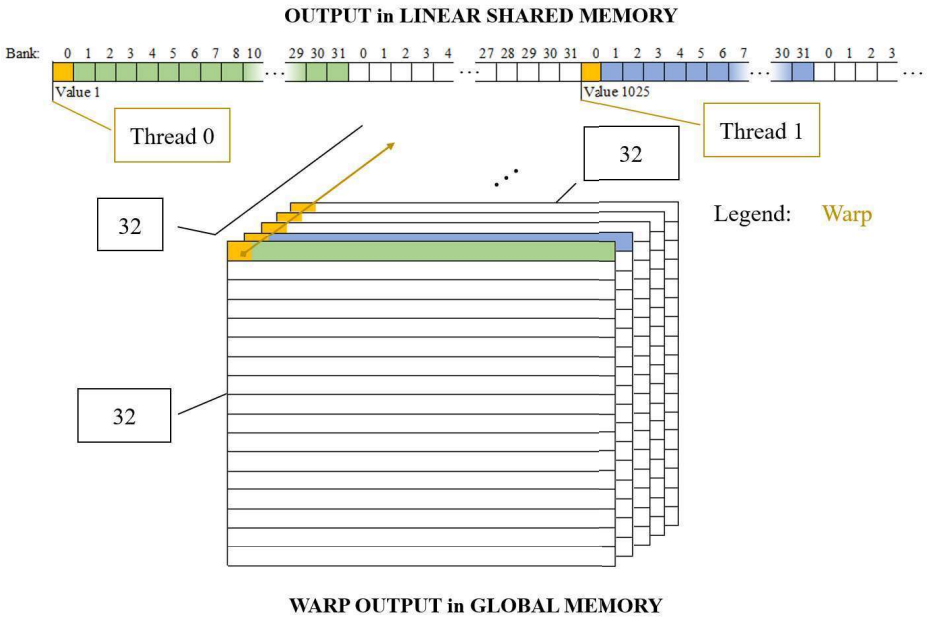

Figure 5.11. Save back operation where bank conflicts are caused by the W innermost shared memory structure.

To avoid these conflicts, the shared memory logical structure and access pattern must be changed to allow for a save pattern that will produce non-conflicting bank access patterns and ideally still be 32 threads wide for optimal throughput. Thus, it was decided to change the K dimension of the shared memory tensor to innermost since it was determined that full warps would always be used in the K dimension of the computation. Doing so allows every thread in a warp to save to one unique bank in shared memory as seen in figure 5.12.

Unfortunately, this change to K innermost causes later access in the save from shared memory to global memory to produce a substantially larger numbers of bank conflicts. In fact, there will be $(T_w * B_w) + S - 1$ conflicts as each w value corresponds to the same bank in shared memory. This would substantially degrade the performance of the save operation since only one thread at a time would be able to perform a save of one single value.

Fortunately, a simple fix can be found by padding the shared memory allocation. The initial shared memory tensor used an exactly 32 wide K innermost dimension. However, by adding one to this dimension at allocation time, the logical bank mapping is changed

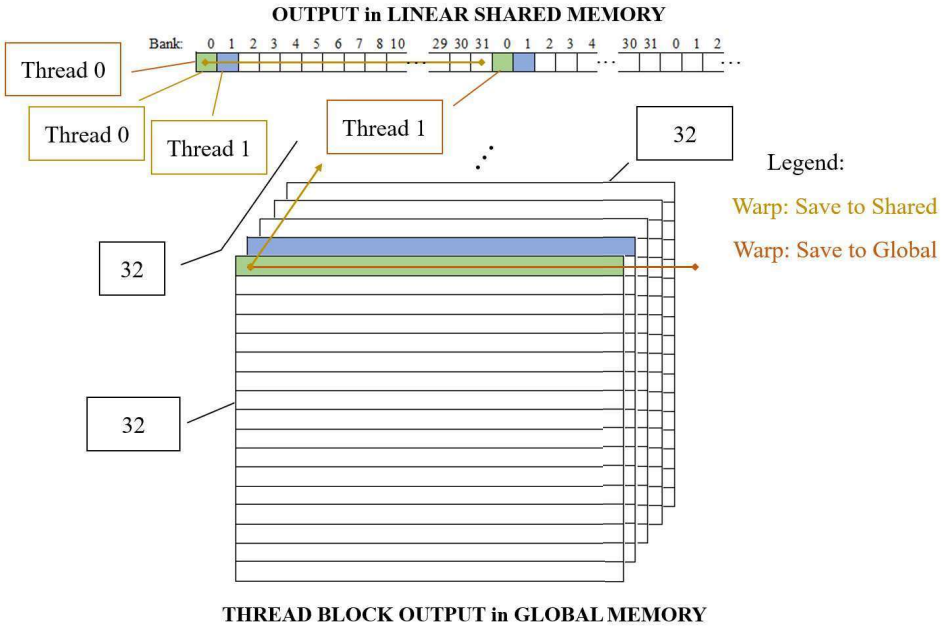

Figure 5.12. Save back operation where bank conflicts are caused by the K innermost shared memory structure.

subtly as seen in figure 5.13. Saves to the shared memory tensor still occur in the exact same manner as previously, but now the adjacent w values will be in separate but adjacent banks. The padding indices of the tensor will take space and be unused, but the slight increase in overall shared memory size is far less important, in most cases, than the concurrency gained by allowing entire warps to participate in single shared memory operations.

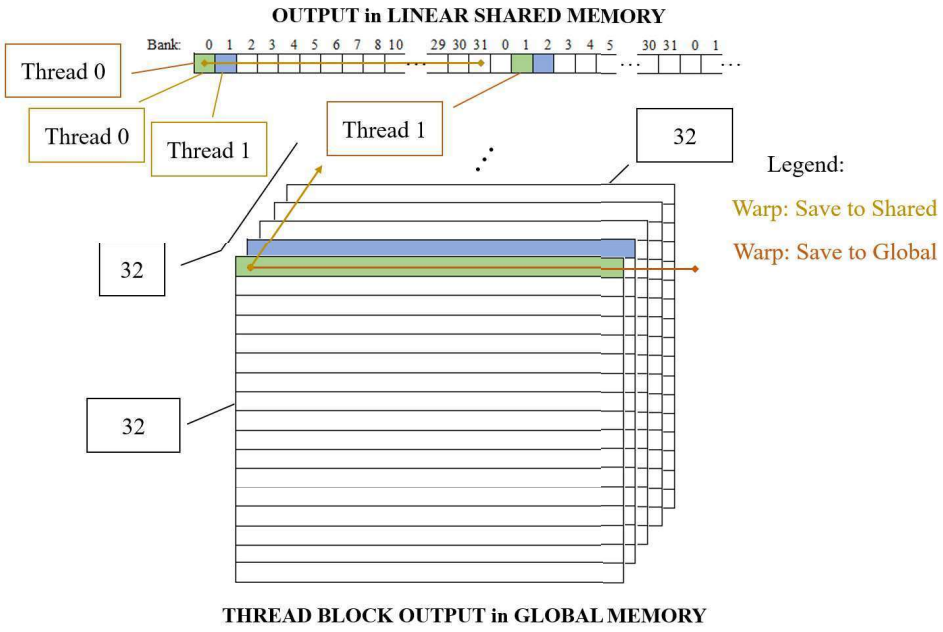

Figure 5.13. Save back operation where bank conflicts are avoided by the K innermost shared memory structure with added padding.

CHAPTER 6

C DIMENSION REDUCTION PATTERNS

Having implemented thread and thread block level reuse patterns, overall performance had improved substantially. However, it was observed during testing that the operational intensity of the kernel would improve as the input problem size became larger. The overall times for larger problems were, of course, slower due to the larger problem size, but the operational intensity showed that the instructions were executed more effectively in that time.

By examining the changes between the execution of these differing problem sizes, it was determined that the larger problem sizes provide more thread blocks for execution than the smaller problems which in turn provide greater opportunities for concurrency. Some of the smaller problem sizes were, in fact, found to not even be using all available SMs for the computation of the problem. Having multiple thread blocks per SM allows the SM to switch out warps from different blocks in execution pipelines when stalls occur. Not having enough thread blocks to even fit all SMs means that not only can the SMs not perform these switches, but some SMs are entirely unused wasting potential concurrency. Therefore, it was decided that additional steps would need to be taken to ensure sufficient blocks were present in each execution to provide multiple thread blocks to each SM.

With tiling already present in the K, H, and W dimensions, and since R and S are quite small in most pipelines, the only remaining dimension to adjust was the C dimension. The C dimension is a reduction dimension. Like R and S, the C dimension is used in the creation of the output tensor, but is not present as a dimension in that output. The values in the C dimension are applied to all values in the output. Since all values of C must be used for each element of the output, the C dimension cannot be split into discrete sets and still result in the complete output values. Thus, complete tiling cannot be performed in this dimension. However, the dimension can still be divided into subsets to better distribute

work to more threads; provided there is some eventual reduction on the resulting values to reach the final output values.

6.1 Theoretical *C* Dimension Division

Dividing the *C* dimension was considered at any of three levels: within a warp, within a thread block, and within the thread block grid. All these options are viable for creating this divided *C* dimension, but the specific requirements of each vary slightly. In every case, creating this divided *C* would increase the total number of thread blocks present in the computation of a given problem size as desired.

The first approach is to divide the *C* dimension between different threads in a warp. In this case, each thread is still responsible for a T_k in the *K* dimension, however, a set of adjacent threads only computes T_k output values for a range of selected *C* values. The number of such groups can be defined as W_c and the number of threads in that group can be defined as W_k. Since a warp is still 32 threads, $W_c * W_k$ must equal 32. This new division of work from input to output can be seen in figure 6.1 and similarly divides the filter set.

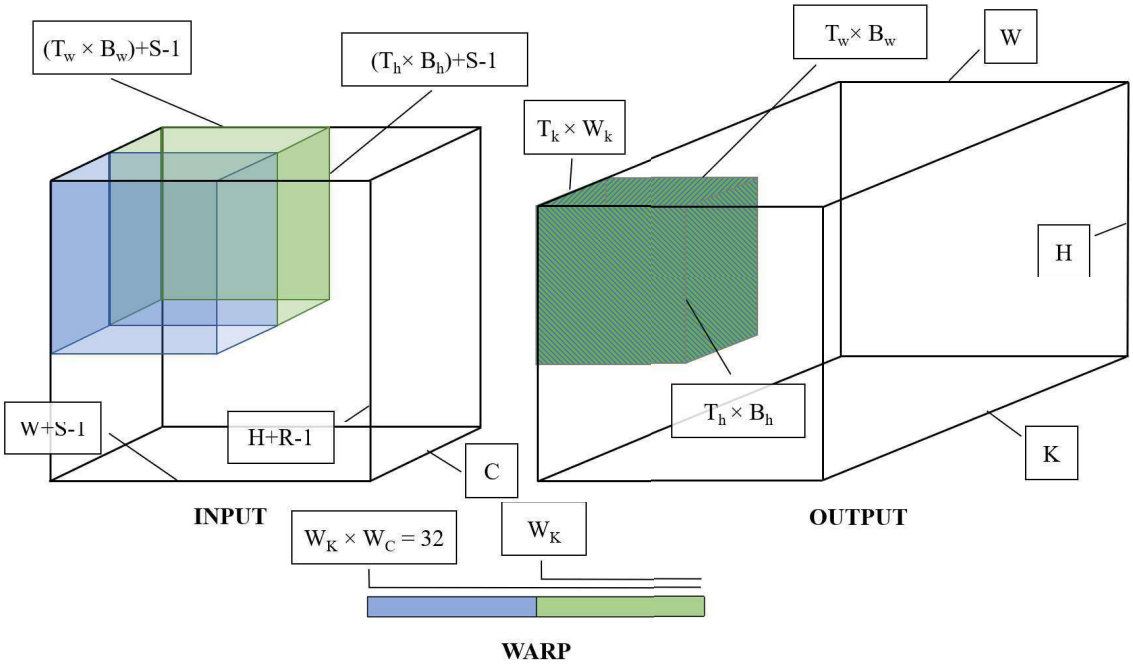

Figure 6.1. C division within a warp and its mapping in input and output tensors.

The second approach is to divide the *C* dimension across different warps in a thread block. In this case, computation from a warp level is almost entirely unchanged, still performing output computations for $T_k * 32$ output planes. The difference is that a warp is responsible for some subset of *C* values that make up those planes. Other warps in the thread are responsible for computing the output values produced by the other subsets of *C*. This problem division scheme can be seen for input and output in figure 6.2 and similarly divides the filter set.

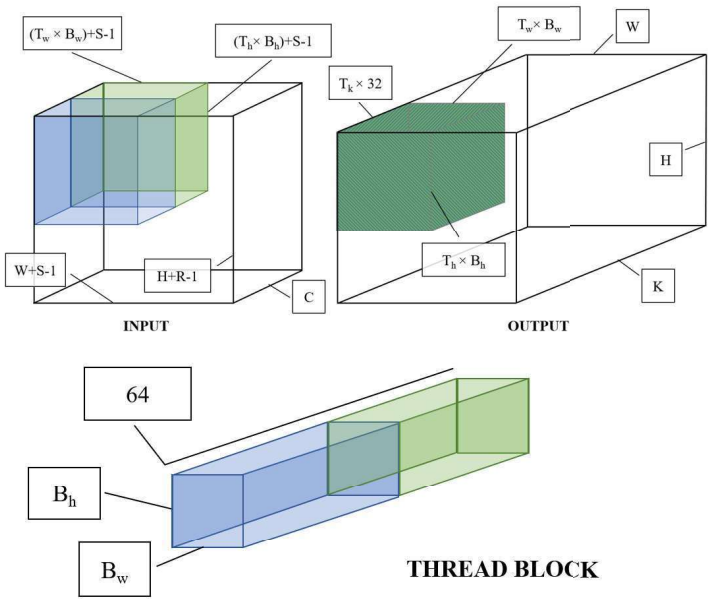

Figure 6.2. C division within a thread block and its mapping in input and output tensors.

The third approach is to instead divide the *C* dimension across different thread blocks in the thread block grid. In this approach, a normal grid spanning the problem space is created, then some multiplier is used to duplicate this grid. Each new copy of the grid is responsible for computing its own output values as normal from some subset of the input values. This final problem layout can be seen in figure 6.3 for input and output.

In each of these cases, the values computed and stored in registers by each thread are only a partial computation of the complete resulting value. As such, some reduction operation needs to be performed between these partial output values to determine each

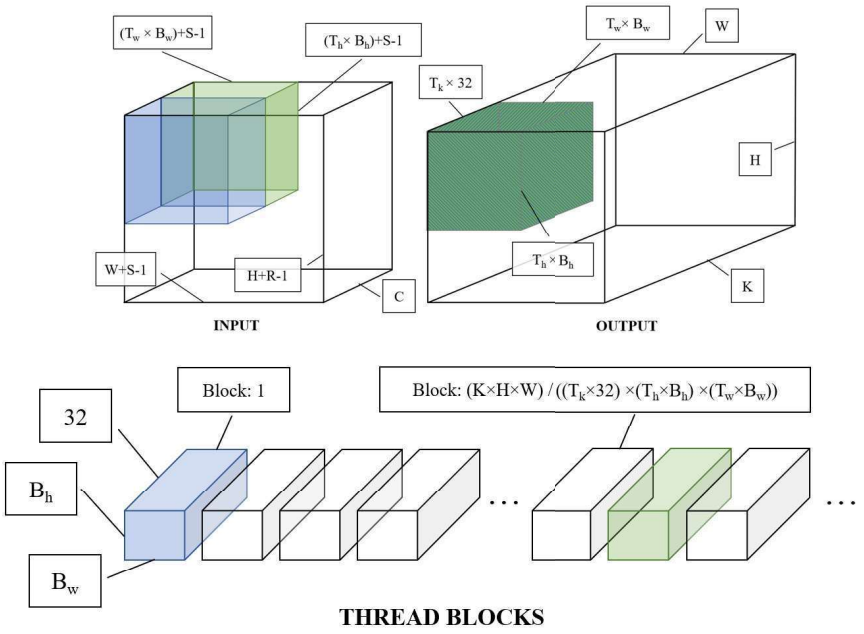

Figure 6.3. C division within a the thread block grid and its mapping in input and output tensors.

final output value. The implementation of this reduction varies between the different division levels.

6.2 Warp, Shared Memory, and Global Division Implementations

Implementing the division and computation portion of the warp level division was done simply by determining a thread's group by its index within the warp one time at the start of the computation cycle. From there, the thread would operate over values between the indexes specified by the thread's group. This would be done by simply precomputing the thread's offset in C and then jumping by the W_c through the outer for loop seen previously in figure 5.8. The calculations for this can be seen in figure 6.4.

The reduction operation for the intra-warp division had several options. Since the reduction would be across registers in a warp, warp shuffling operations could be used to implement the reduction. Using warp shuffles (which are warp level reductions only involving registers), the values could be reduced to a single warp group's registers without

```
int cGroupWarp = threadIndexWarp / W_k;
for(cJumping = 0; cJumping < C; cJumping+=W_c) {
      oneKern = Kernel[ (cJumping + cGroupWarp)*R*S*K + (r)*S*K + (s)*K + (kIndex + kGlobal) ];
      …
```

Figure 6.4. C warp division implementation.

passing the values to shared or global memory. From there, the now complete values found in the first warp group's registers could be saved back to global memory in the usual manner.

Unfortunately, since the result values would only be found in some subset of the warp, the previously discussed changes to the save operation to avoid bank conflicts would not be possible without adding substantial amounts of padding. However, with the shorter depth spanned by each warp in the *K* dimension, additional blocks would be required in the computation of a given problem. This in turn allowed for better parallelization on hardware.

In the case of the intra-thread block division, the principle for execution was similar to that of the warp level division. Different warps were identified as being a part of separate groups based on their linear thread index and then set to work on portions of the *C* dimension accordingly. Again, the actual computation limits were facilitated by simply adjusting the bounds on the outer *C* loop, keeping indexing simple and fast. The grouping calculations can be seen in figure 6.5.

```
int cChunk = threadIdx.y / B_w;
int cStart = cChunk * CDivisionSize;
int cBound = cStart + CDivisionSize;
for (c = cStart; c < cBound; c++) {
  …
```

Figure 6.5. C intra-thread block division implementation.

However, in this case, the partial values in registers were separated across different warps so warp shuffling would not be possible. Nevertheless, since all of the warps were part of the same thread block, shared memory level atomics could be used to collect the partial values into shared memory prior to their movement back to global memory.

Since these shared memory level operations would again use a full warp, the normal save structure could be used.

Finally, in the case of inter-thread block division of *C*, indexing would be determined and executed in a similar way to the previous versions. It would differ only in that instead of being divided by thread index it would instead be divided by linear block index. This computation is seen in figure 6.6. The major differences in this version appear in the save back operation. Upon a thread block's completion, registers in that thread block only contain a partial solution. Since shared memory is only shared within thread blocks, shared memory atomics cannot be used. Instead, global memory atomics were used for the reduction of the output values.

```
int blockIdxXAdjusted = blockIdx.x % BlockXCoverageNumber;
int kIndex = blockIdxXAdjusted * blockDim.x * T_k + threadIdx.x;
int cStart = (blockIdx.x / BlockXCoverageNumber) * CDivisionSize;
int cBound = cStart + CDivisionSize;
for (c = cStart; c < cBound; c++) {
  …
```

Figure 6.6. C inter-threadblock division implementation.

By reducing in global memory, the previously discussed save operation could be used. However, the atomic save to global memory means that warps can collide while saving and in turn cause stalls. Due to the memory structure of the GPU, this was essentially the only option for the reduction of the partial result values.

6.3 Warp Division Complications and Resulting Optimal

After implementing each of the three discussed versions, it was discovered in testing that the optimal design was the inter-thread block division. This was surprising behavior considering the global reduction operation found in this implementation. However, upon careful examination, the reasons for this performance difference became clearer.

In the case of the division within warps, the performance was generally much worse than the performance of the thread block division, even when both implemented with the same save structure. The performance discrepancy was found to be largely due to the change in the output reach obtained by a given warp. Since the warp division worked by

separating different portions of the warp to different C chunks, the number of output K planes generated by the warp in its execution was reduced.

The reduction in K reach of each warp is the intended behavior and allows for additional thread blocks to be created to cover the problem space. However, the unfortunate side-effect of this change is that now the reuse of a given input plane is decreased. The reuse of a loaded input plane in the case of a full warp without division across C is $32 * T_k$ output planes. Every thread in the warp can use the information loaded for each of its planes contained in registers. In the case of the warp division though, the amount of reuse for a plane becomes $W_k * T_k$, where $W_k * W_c = 32$. Thus, the reuse of the input is reduced by W_c times.

Overall, this means that for a given problem size, the same input elements will have to be loaded from global memory more times in separate thread blocks. A graphical version of this behavior can be seen in figure 6.7. This increase in the number of loads from input makes this approach perform worse relative to the global method and thus hid the performance gains from the increased concurrency.

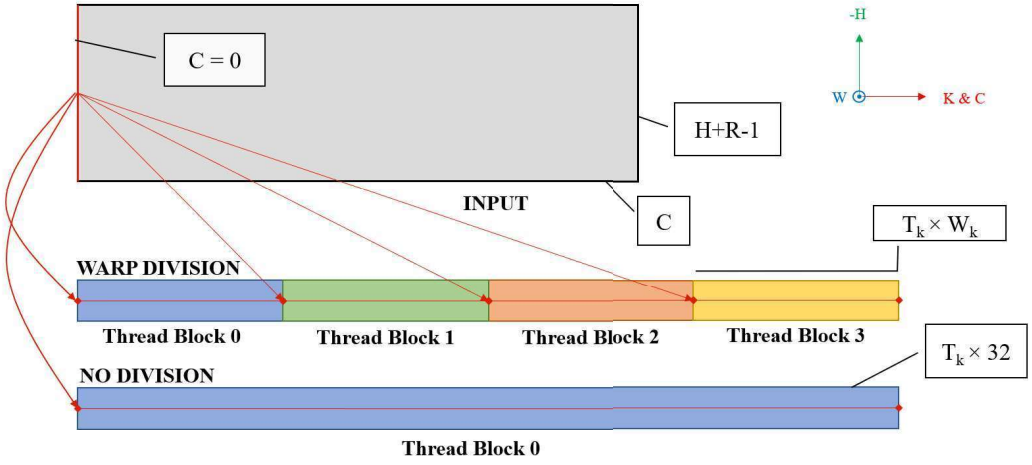

Figure 6.7. Input plane reuse over a given output distance in K for the warp-divided C and the non-divided C versions of the kernel.

In the case of the division across warps inside a thread block, the performance relative

to the global reduction pattern is also lower. This method does not suffer from the same loss of *K* dimension reuse. Instead, the duplicate warps in a block that are allocated to the same planes in output mean that less warps can be allocated to other portions of output in *H* and *W*. As seen in figure 6.8, for the same thread block size (same number of total threads), the thread block reaches less elements in both *H* and *W* relative to the standard version. As discussed, there is reuse present in the *H* and *W* dimensions and so losing this reuse is degradative for performance. Also, thread blocks are only added from a reduction in these dimensions. Therefore, large thread blocks that can still reach the same space in *H* and *W* as in the original do not produce the desired concurrency gains.

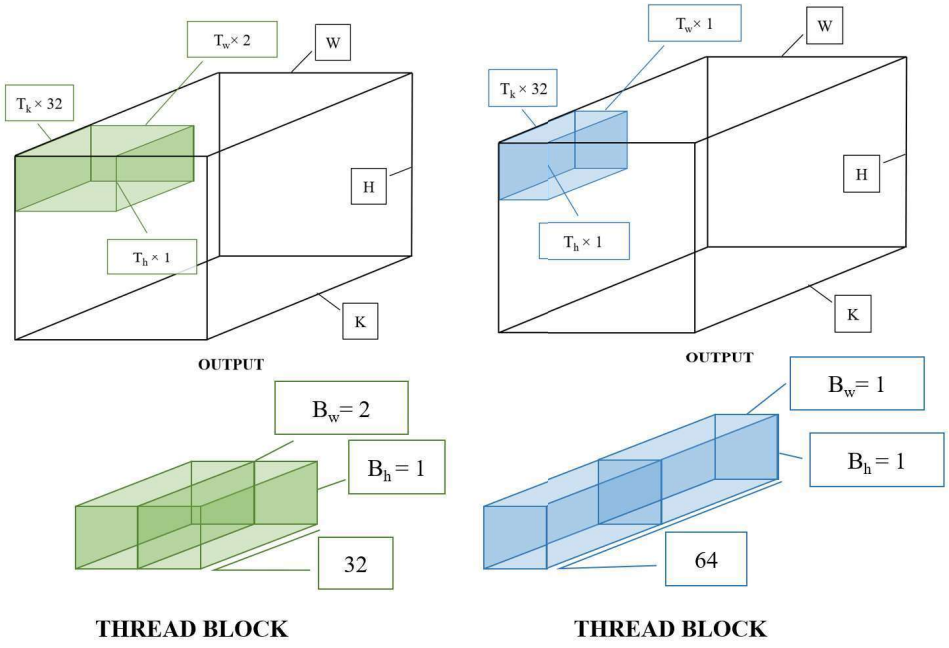

Figure 6.8. Thread blocks with two warps without C division (left) and with C division (right) and their mappings to global output.

Essentially, the separation of *C* across different thread blocks allows for each block to behave as it had been designed previously but still adds to the total number of thread blocks. The only difference is that a thread block will work on a reduced version of the problem where the *C* dimension is some fraction of the true dimension. The cost of this approach is that any saves to the same locations in global memory will have to be serialized

and execute the division amount of times. However, in comparison to the amount of work done in computation, the number of total saves is still sufficiently small to gain an overall performance increase thanks to the increase in concurrency between separate thread blocks. One other small change to the program assumptions is required due to this atomic reduction into global memory. It is now required for output global memory to be zeroed prior to execution so that results will correctly accumulate there.

CHAPTER 7

FILTER LOAD

After the numerous other modifications, the only remaining unmodified access to any structure was the load of filter elements necessary for computation. This load had continued to remain unchanged from its single element direct global memory load format found in the initial register only version. The reason this load remained unchanged was due to its location in the code's loop structure and the limited reuse possible for each element in the filter set.

Each item in the filter set can only be reused in the H and W dimensions of a single output plane. This is due to the filter set being indexed by K, C, R and S. Therefore, a single filter element will be used only within one input plane C value to produce one output plane K value for all H and W elements where the filter would fit in accordance with the dimension of the filter plane. This can be viewed more clearly in figure 7.1. Since the H and W dimensions of both input and output were subdivided into tiles and in turn split between thread blocks, the potential for reuse in a thread block was even further limited.

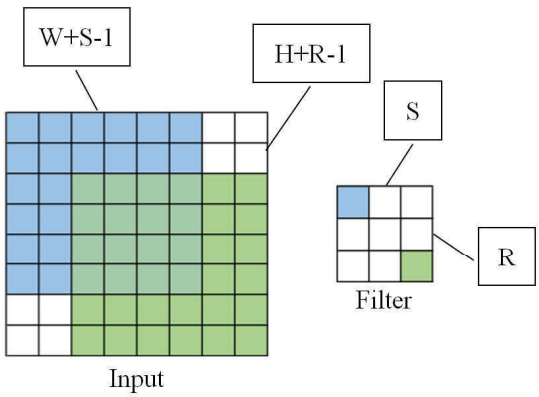

Figure 7.1. Maximum filter element reuse in the input plane where $K = k$ and $C = c$.

From an implementation perspective, changes to this load would be difficult as well. Since the load was found inside several loops, any load to shared memory would require barriers inside the loops to make sure any shared loads were completed before moving on to computation. In experimentation, this increase in the number of thread barriers was found to be very degradative to performance. Tests were also made to use a technique called double buffering to avoid these barrier delays, but this did not succeed in resolving the issue.

Fortunately, by carefully examining the current load structure as translated to assembly, it was found that with a sufficiently large thread H and W tile, the load time for a given filter element could be covered by the computation operations executed on the previously loaded element. Since these two operations used different hardware pipes, this allowed for concurrency to cover this global load almost entirely. Unfortunately, this also meant that if the tile H and W became too small the concurrency would not be sufficient to hide the loads and would cause warp stalls. This was found not to be a problem in most cases since this only occurred with very small tile sizes (around 15 total elements) which could generally be avoided.

CHAPTER 8

FINDING THE OPTIMAL PARAMETER VALUES

Having structured the code in such a way as to allow for parameters in various tile and block sizes, the next challenge was finding an optimal set of dimensions for a given problem size. Due to various constraints placed on the problem by assumption and design choice, the total number of options for a given problem had been reduce substantially. However there could still be up to many thousands of possible combinations. Therefore, considerations would have to be made about the best way to approach finding the best of these combinations.

8.1 Calculated Values

The first approach considered was to use the models and equations determined during the creation of various design levels of the code to indicate potential performance. Values for thread level operational intensity, block level operational intensity, occupancy, blocks per SM, warps per block, and so on could easily be precomputed using simple scripts. Traversing the combination space and computing these values was extremely fast and only required known inputs like the hardware specifications and problem sizes.

The challenge was evaluating the resulting combinations of parameters. Although all of the previously mentioned calculated values are indicative of performance, they are not the actual performance. It was also found to be very difficult to model execution concurrency with greater accuracy than values such as the maximum number of blocks per SM. Where concurrency is such an important feature of GPU execution and was such an important consideration in the code design, it was unfortunate that more time was not available to attempt to more correctly model this aspect of the execution. However, even without the most optimal modeling equations, determining balance and fairly performant combinations could be done from the resulting computations by manual inspection.

8.2 Searching the Problem Space

The second approach tested sought to better automate the selection process and most accurately predict the performance of a given combination. In this approach different sizing parameters were simply tried on the actual kernel and the execution time was measured. This was done for all valid combinations in an automated script and the lowest time could then be selected. This not only assured the optimal or near optimal combination for the design on a given piece of hardware with a given problem, but also could be done in a not unreasonable amount of time.

Since the constrained option space was, for most problems, only in the thousands at maximum with a very short execution time, evaluating all combinations could be done relatively quickly. In the case where the number of combinations could be up into the thousands, total trial time could be around an hour. For more edge case problems with certain very small dimensions, the total time could be as little as fifteen to twenty minutes. Considering the level of certainty this approach gave to the performance of a specific combination, this was a good tradeoff.

Also, as inelegant as trying all combinations was, it was still only a small subset of the true design space due to the applied constraints. As well, in a use case these evaluations would be done on the different levels of a given CNN pipe only once prior to training and or use. After evaluation was complete and optimal tile sizes were selected, the given kernels could be run any number of times for any amount of time with the best possible performance offered by this kernel design. Not only that, but this approach would be possible for any piece of hardware for any given problem, satisfying the adaptability goals desired for this kernel.

CHAPTER 9

RESULTS RELATIVE TO CUDNN AND TVM

To best evaluate the performance of this kernel, timing tests were performed on problem sizes for two common CNN pipelines. The ResNet and Yolo pipelines are both common use cases that feature many of the different problem sizes and execution patterns for which such a kernel would be employed [6], [12]. As such, they provide a reasonable testbed for performance evaluation. The convolutional layers of these two networks are shown in table 9.1.

To better understand the timing results, timing tests were also conducted on the given pipelines using the cuDNN and TVM kernels. cuDNN is the Nvidia kernel set used for the evaluation of convolution problems. It makes use of both GEMM and domain space transformation techniques in the evaluation of problems [4]. However, results for this comparison would be relative to only the GEMM cuDNN kernels to better compare this kernel with kernels having the same minimum number of operations. The TVM kernel is a commonly used GEMM solver for the convolution problem that employs autotuning to select its kernel configuration [3].

Timing these two kernels would provide a good comparison by which to evaluate the effectiveness of this newly created kernel. Actual timing was conducted using Cuda's build in timing events [10]. These events would be used to indicate the start and end of kernel run and thus were used to log the individual runtimes. Because of minor fluctuations in the runtimes, kernels were run 24 individual times. Between runs, caches were cleared, and in the case of the created kernel the output was reset.

These successive tests were conducted on the same GPU for each kernel set, but two different devices were used to test the mobility of the created kernel between different hardware architectures. The first used was a Nvidia 2080 Ti GPU paired with an AMD Ryzen Threadripper 3990X 64-Core CPU running Ubuntu 20.04. This GPU was the type

Layer	K	C	H/W	R/S
Y0	32	3	544	3
Y2	64	32	272	3
Y4	128	64	136	3
Y5	64	128	136	1
Y8	256	128	68	3
Y9	128	256	68	1
Y12	512	256	34	3
Y13	256	512	34	1
Y18	1024	512	17	3
Y19	512	1024	17	1

Layer	K	C	H/W	R/S
R1*	64	3	224	7
R2	64	64	56	3
R3	64	64	56	1
R4*	128	64	56	3
R5*	128	64	56	1
R6	128	128	28	3
R7*	256	128	28	3
R8	256	128	28	1
R9	256	256	14	3
R10*	512	512	14	3
R11*	512	256	14	1
R12	512	512	7	3

Table 9.1. Convolution layers for Yolo (left) and ResNet (right) [* indicates stride 2 layers].

of GPU primarily used in the development of the kernel. A Nvidia V100 GPU paired with an Intel(R) Xeon(R) CPUE5-2680 v4 26 core CPU running CentOS 7.8.2003 was the other device on which tests were conducted. Evaluating times on both of these hardware architectures allows for some additional insight into the ability this new kernel has to be used in the general case. The median of the resulting times for each set of hardware and for each kernel set can be seen in figures 9.1 - 9.4. The runtime differences between some of the kernels between architectures can be seen in figures 9.5 and 9.6. In these figures, the times for the approach discussed in this thesis are labeled CSG standing for ”C Split Globally” indicating the thread grid level C split.

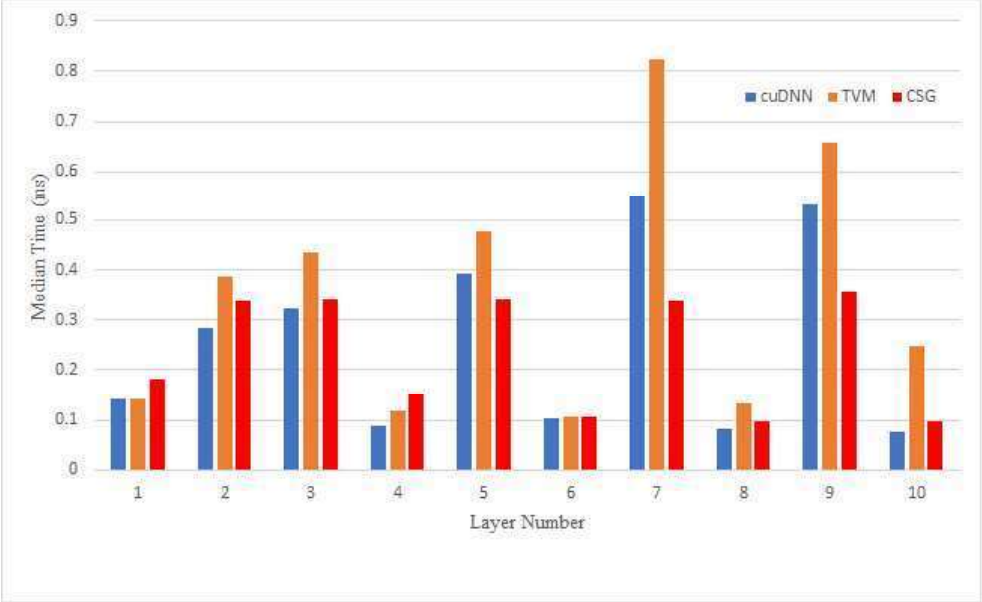

Figure 9.1. Yolo performance on ti2080.

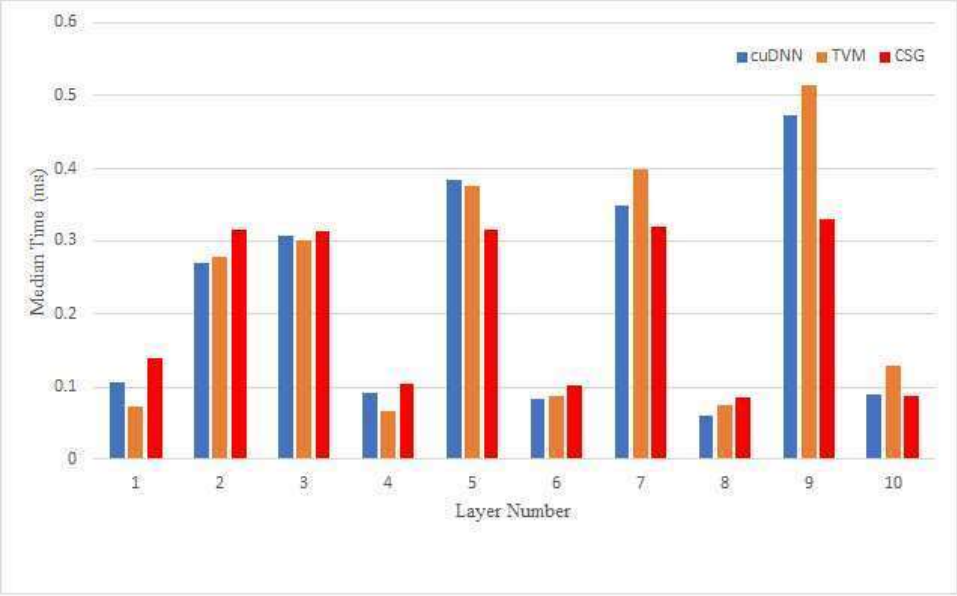

Figure 9.2. Yolo performance on v100.

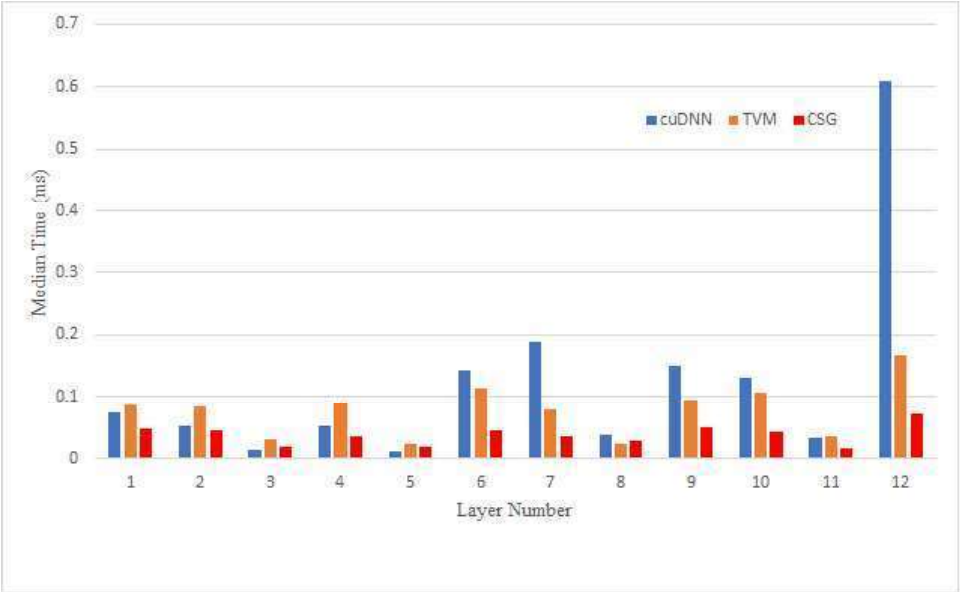

Figure 9.3. ResNet performance on ti2080.

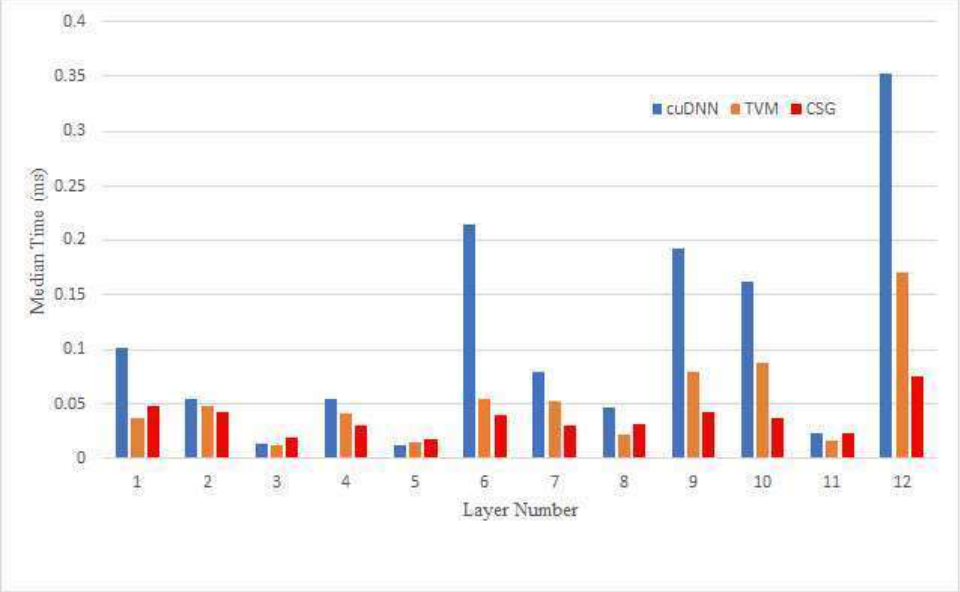

Figure 9.4. ResNet performance on v100.

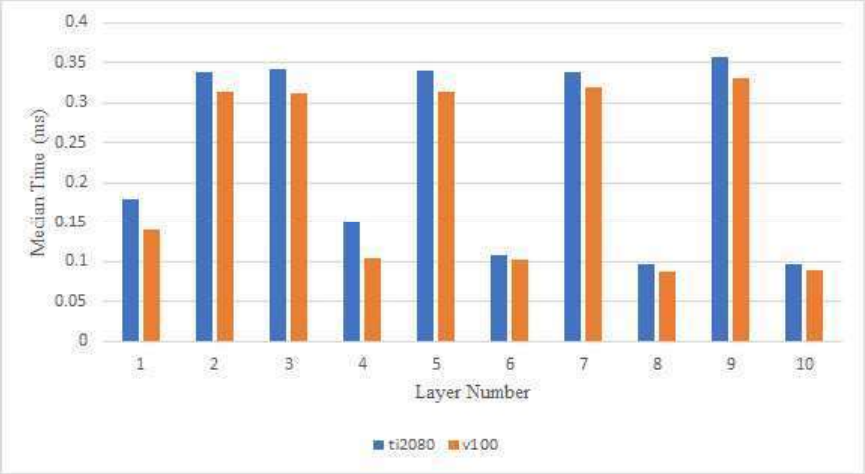

Figure 9.5. Yolo architectural performance difference for the developed kernel.

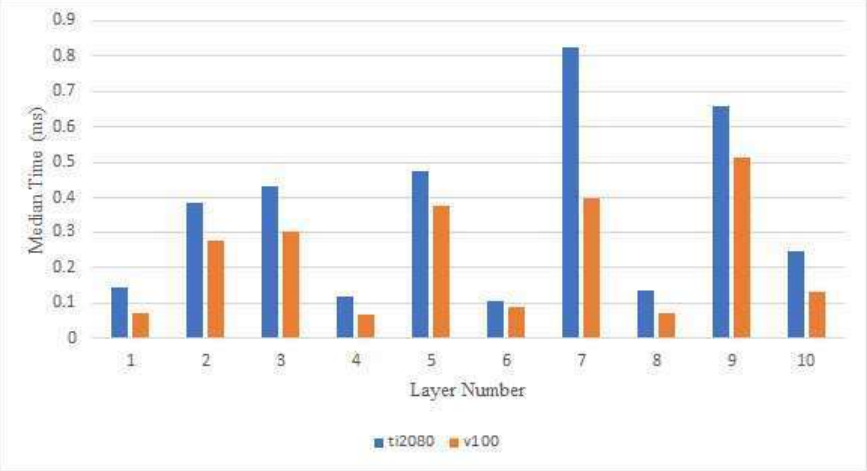

Figure 9.6. Yolo architectural performance difference for TVM.

CHAPTER 10

CONCLUSIONS

Given these timing values, it can be seen that there are several cases where the resulting times for the newly created kernel are both better than cuDNN and TVM. However, there are also many cases where the times have not improved. Overall, though, one can observe that in most cases the created kernel performs better or close to the TVM kernel. On the other hand, the performance relative to the cuDNN kernels can vary widely depending on the problem.

This kernel's performance is most apparently worse than the cuDNN and the TVM approaches on the layers where the filter size is 1x1. Examples of this are seen in layers 3, 5, 8, and 11 in the ResNet pipeline and layers 4, 6, 8, and 10 in the Yolo pipeline. These problem sizes are unique because of their 1x1 filter size. In this problem size, the evaluation pattern essentially becomes that of repeated matrix multiply. As such, the GEMM implementations have a clear advantage where they are already simply a modified form of matrix multiply.

Alternatively, the performance for the stride two problems seen in layers 1, 4, 7, and 10 in the ResNet kernel are more performant than both the cuDNN and TVM versions. This could be due to the better reuse found in directly approaching the convolution pattern as performed in this kernel verses the more indirect GEMM approach. However, even in the non-stride two cases, performance for the created kernel is better for the mid to smaller problem sizes where it can make the best use of the more equally sized K, C, and H, W dimensions.

As a final note, anyone familiar with the Yolo might notice that the presented graphs do not incorporate its eleventh convolutional layer. The final layer for Yolo is a large operation on a problem with dimension 28269 in K. Since the K dimension is not divisible by 32, the kernel created in this research will not correctly function for this problem size. Divisibility

by 32 was specified as one of the assumptions at the beginning of this kernel's development so no account for another size was taken during development. As such, this layer was not included for testing or comparison.

CHAPTER 11

CONTINUING WORK

To improve the performance of this kernel further, many aspects of this research could be refined. One goal of such efforts would be to further develop the kernel to solve the limiting aspects of the current design. The second goal of the continuing work would be to streamline the preparation that must be done for each specified problem and architecture.

11.1 Kernel Refinement

The kernel contains many areas of further exploration. First among these would be to handle a problem discovered at the end of this research. As discussed, the current approach uses a register set to contain the input tiles required for each thread. This was done to increase the number of operations that could be done for each input element loaded by retaining those values in the fastest memory type possible. However, upon examining the assembled code, it was found that the compiler had removed this storage. Instead, it opts to only move one row of input from shared memory to registers at a time.

Unfortunately, the execution pattern that accompanied this change caused the kernel to load and reload these values to registers multiple times throughout the complete execution. This severely reduces the operational intensity from the theoretical values anticipated for the register level operations. As such, further efforts would need to be taken to develop the code in such a way as to ensure the compiler will generate code in the desired way, or to simply change the approach to account for the compiler's behavior. If a new approach was to be developed, it would follow similar development paths as the original code, but with the compiler's limitations in mind.

Besides this, the filter load directly from global memory was another problem that was only worked upon for a limited amount of time in this research. Several of the issues present in changing filter load into a similar pattern as input and output have already been discussed. Different approaches would need to be explored to find if some alternate

solution could eliminate this limitation.

Continuing, the current kernel designs all used a specific approach to the tiling structure in the K dimension. All operations in this dimension would have to be done in some multiple of 32 since this was the orientation of warps in a given thread block. However, warps could be spilt into more dimensions like the thread blocks. This would allow for numerous additional options in both load patterns and tile sizes for the entire problem. However, it would add additional complexity to the kernel design since threads in a warp would have more to track than in their simple linear structure found in the current kernel.

Another area worth exploring further would be the use of an additional kernel for the 1x1 filter sizes. As discussed, these filters result in what is essentially a matrix multiply. As such, a more direct approach to this matrix multiply problem could potentially shore up these far less performant problem sizes. This could simply be added to the decision program's options to select depending on the problem size. This approach is already taken with the stride 2 operations that use a slightly modified version of the discussed kernel.

Several assumptions were made for this problem from the start. One of these was that any given problem would only have a batch size of one. This translated to only a three-dimensional input and output for any given layer. However, during training, it is common for images to be given to the CNN pipe in batches. This translates into a four-dimensional input tensor with another four-dimensional output tensor at each layer. These extra dimensions add numerous other opportunities for a kernel to leverage potential reuse, division of work, and overall utilization of the GPU for small image sizes. However, such changes might result in a lower performance for the single image case such as is common when the CNN has completed training and is in use. The original kernel could still be used for these situations to eliminate this potential problem.

11.2 Parameter Selection and Hardware Variations

Besides changes to the kernel, additional changes might also be made to the parameters. It was asserted in the design of the parameter selection scripts that the tiles sizes selected would always exactly fit the output size. It was also assumed that the input was padded to exactly match the output size based on the filter size. However, this was an imposed limitation and could be changed. There is the potential that by adding additional

padding to the input prior to computation that better tile sizes could be selected for the problem. This would in turn add more overall operations, but there is the potential that the better tile structure would have a greater impact on performance.

Along with this, since the tile sizes were always forced to exactly fit, no allowance was made for any sort of scheme that left partial tiles. Allowing partial tiles would again provide greater choice when selected tile size for a given problem, but would also require changes to the kernel. The benefits would be similar to the additional padding option discussed, but would not require the padding operations. The potential downside to this approach would be that any sort of scheme allowing partial tiles would require greater complexity in the kernel, and could have very un-performant partial tiles that could hinder overall performance.

Continuing with tiles, as discussed, the final approach selected for determining the best tile size was a simple exhaustive search of the tile option space. Although this was not terribly time-consuming, it could certainly be improved. One option for this would be to refine the computational approach that was used previously to better precompute a given parameter set's operational behavior, including parallelism. These values could then be used in the training of a machine learning algorithm or perhaps something as simple as an approximate Q learning algorithm. With such a trained algorithm, selection of parameters for a new problem size or architecture could be done quickly by pre-computing the indicative values and then passing them through the trained algorithm. A mixed approach could even try several of the results determined to be best by the algorithm and then pick the best option. This could potentially keep the performance obtained by an exhaustive search while also greatly reducing the decision time.

Additionally, all of the tests conducted in this research were performed on Nvidia GPU's due to their availability. As such, the potential for this kernel was not fully explored. Its relative performance to the Nvidia cuDNN kernels could change, perhaps substantially, on GPUs that are not also produced by Nvidia. Since this kernel focuses on adaptability and basic GPU behaviors, it could potentially maintain its performance between GPUs produced by different companies. As such, this would be an important area of further testing to determine the current capabilities of the kernel and better understand its current limitations.

Finally, one other remaining area of study would be to apply the systems used to develop this kernel with frequency domain convolution approaches. Although the mathematical techniques would be different than those found in this kernel, use of parameterized tiles might still be beneficial to improving performance. The addition of other kernel types would provide even more options to a selection script and potentially allow for greater adaptation to a given problem and hardware.

11.3 Concluding Thoughts

Although in its current form, the kernel is not always successful at achieving higher performance than other approaches to convolution, with further development there is a great possibility for additional gains to be made. Numerous paths of research are still open for further exploration. The kernel's ability to adapt to different problems has already been demonstrated and this aspect of the kernel still has potential room for improvement. This approach to the convolution problem has shown that it has the ability to yield higher performance than alternative methods in specific cases. Thus, this research has provided an initial exploration into the design approach, on which future improvements will hopefully be founded.

REFERENCES

[1] Albawi, S., Mohammed, T. A., and Al-Zawi, S. Understanding of a convolutional neural network. In *The International Conference on Engineering and Technology 2017* (2017).

[2] Balaprakash, P., Dongarra, J., Gamblin, T., Hall, M., Hollingsworth, J. K., Norris, B., and Vuduc, R. Autotuning in high performance computing applications. *Proceeding of the IEEE 106*, 11 (July 2018).

[3] Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Cowan, M., Shen, H., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. Tvm: An automatedend-to-end optimizing compiler for deep learning. In *USENIX Symposium on Operating Systems Design and Implementation* (2018), pp. 578–594.

[4] Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. cudnn: Efficient primitives for deep learning. In *arXiv:1410.0759* (2014).

[5] Gupta, M. Cutlass convolution. https://github.com/NVIDIA/cutlass/blob/v2.4.0/media/docs/implicit_gemm_convolution.md, 2020.

[6] He, K., Zhang, X., Ren, S., and Sun, J. Identity mappings in deep residual networks. In *European conference on computer vision* (2016), p. 630–645.

[7] Jorda, M., Valero-Lara, P., and Pena, A. J. Performance evaluation of cudnn convolution algorithms on nvidia volta gpus. *IEEE Access 7* (2019).

[8] Lavin, A., and Gray, S. Fast algorithms for convolutional neural networks. *Arxiv* (2015).

[9] Li, R., Xu, Y., Sukumaran-Rajam, A., Rountev, A., and Sadayappan, P. Analytical characterization and design space exploration for optimization of cnns. In *ASPLOS '21* (Apr. 2021).

[10] NVIDIA. *CUDA C++ Programming Guide*. Santa Clara, CA, USA, Dec. 2020.

[11] Pratt, H., Williams, B., Coenen, F., and Zheng, Y. Fcnn: Fourier convolutional neural networks. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases* (Dec. 2017).

[12] Redmon, J., and Farhadi, A. Yolo9000: better, faster, stronger. In *IEEE Conference on Computer Vision and Pattern Recognition* (2017), p. 7263–7271.

[13] Russell, S. J., and Norvig, P. *Artificial Intelligence A Modern Approach*, third ed. Pearson Education, New York City, New York, 2018.