Technical Report

Memory-Aware Fusing and Tiling of Neural Networks for Accelerated Edge Inference

Jackson Farley
Andreas Gerstlauer

UT-CERC-21-01

May 7, 2021

Computer Engineering Research Center
Department of Electrical & Computer Engineering
The University of Texas at Austin

2501 Speedway, Stop C8800
Austin, Texas 78712-1234

Telephone: 512-471-8000
Fax: 512-471-8967

http://www.cerc.utexas.edu

The University of Texas at Austin
Electrical and Computer Engineering
Cockrell School of Engineering

Memory-Aware Fusing and Tiling of Neural Networks for Accelerated Edge Inference

Jackson Farley, Andreas Gerstlauer
Electrical and Computer Engineering
The University of Texas at Austin

Abstract

A rising research challenge is running costly machine learning (ML) networks locally on resource-constrained edge devices. ML networks with large convolutional layers can easily exceed available memory, increasing latency due to excessive swapping. Previous memory reduction techniques such as pruning and quantization reduce model accuracy and often require retraining. Alternatively, distributed methods partition the convolutions into equivalent smaller sub-computations, but the implementations introduce communication costs and require a network of devices. However, a distributed partitioning approach can also be used to run in a reduced memory footprint on a single device by subdividing the network into smaller operations.

This report extends prior work on distributed partitioning using tiling and fusing of convolutional layers into a memory-aware execution on a single device. Our approach extends prior fusing strategies to allow for two groups of convolutional layers that are fused and tiled independently. This approach reduces overhead via data reuse, and reduces the memory footprint further.

We also propose a memory usage predictor coupled with a search algorithm to provide fusing and tiling configurations for an arbitrary set of convolutional layers.

When applied to the YOLOv2 object detection network, results show that our approach can run in less than half the memory, and with a speedup of up to 2.78 under severe memory constraints. Additionally, our algorithm will return a configuration with a latency that is within 6% of the best latency measured in a manual search.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

There has been a proliferation of complex machine learning (ML) problems in edge applications. As pointed out by [8], running ML applications on the edge can increase privacy, improve latency, reduce cloud communication, and require less energy. However, most state-of-the-art ML networks have significant memory requirements that can exceed available memory on a resource-constrained edge device. Even with virtual memory enabled, exceeding memory bounds comes with severe latency penalties due to excessive swapping between memory and disk. As a result, it is a significant challenge to run networks locally on an edge device.

Numerous commonly used neural networks contain a series of convolutional layers to process image data. Many convolutional layers, especially layers earlier in the network are feature-heavy, with a large amount of memory needed for inputs and outputs. Previous approaches to reduce memory footprints of neural networks such as pruning [1], [5] and quantization [3], [6], [10] modify the network model, require re-training, and experience accuracy degradation. Meanwhile, distributed solutions such as [7] and [13] rely on partitioning convolutions into separate tasks and running them on separate devices,

but they require additional communication and a network of devices. However, such approaches can also be used to reduce the memory footprint of a computation locally on a single device.

In this report, we extend the fused tile partitioning (FTP) approach outlined in [13] to present a memory-aware fusing and tiling (MAFAT) strategy for the execution of large feature-dominated early stages of convolutional neural networks (CNNs) on a single resource-constrained edge device. The FTP approach from [13] combines all layers into one large layer group and fuses them all together in order to reduce communication. By contrast, MAFAT creates two smaller layer groups and tiles and fuses them separately. The smaller fusings and different tilings resulting from more layer groups can reduce the maximum memory footprint of a process. We also develop a model to predict the maximum memory usage of a given MAFAT configuration. Finally, using this predictor, we propose a search algorithm that uses this predictor to return an optimized MAFAT configuration that fits within the provided memory requirement.

Results of applying our approach to a CNN used for object detection [9] show that MAFAT configurations can provide a speedup of up to 2.78 over the original model in tighter memory constraints. Furthermore, our search algorithm returns a configuration with a latency that is within 6 percent of the best measured latency for any configuration.

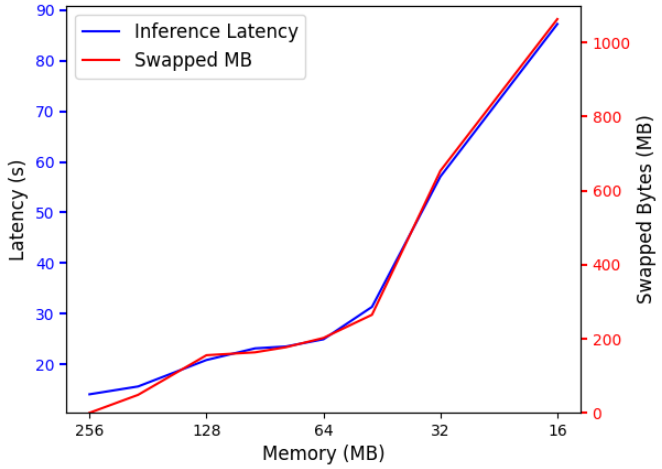

Figure 1.1: The original YOLOv2 implementation for varying memory constraints.

1.1 Motivational Example

Figure 1.1 depicts the latency and number of swapped bytes versus a decreasing memory constraint from running the first 16 layers YOLOv2 [9] on a Raspberry Pi3. The first 16 layers of the network are used because they are the most feature-heavy and present the greatest feature challenge to memory. The measurements and methods used to construct the graph are described in Section 4.2.

Figure 1.1 shows a significant increase in the latency of an inference at tighter memory constraints. It reveals that the CNN exceeds memory constraints at over 192 MB. Once the program goes over memory, the Operating System must swap data between the memory and disk. This swapping pro-

cess has a demonstrated adverse affect on latency. As the memory constraints continue to shrink, the inference latency increases dramatically, with a 16MB memory constraint over 6.5× slower than the original. This motivates a need for optimizations as presented in this report to reduce the latency overhead due to swapping.

1.2 Related Work

The primary approaches to reduce memory on a single device are pruning and quantization. Pruning attempts to remove a portion of the model, such as weights in a filter, but this can result in asymmetric computations that can be difficult to implement [1]. Entire filters can be removed, too, such as in [5]. In both of these cases, pruning severely degrades accuracy and expensive retraining is required afterwards. Quantization of a CNN [3], [6] reduces the number of bits necessary to store weights. Similarly, retraining is often needed to get better accuracy [10]. Quantization also removes model information, i.e. it also degrades the accuracy of the model. By contrast, MAFAT is able to preserve model accuracy while decreasing the memory footprint.

MAFAT is orthogonal to both pruning and quantization. Because the model is preserved, MAFAT can easily be applied to a pruned or quantized network. Combinations of MAFAT and quantization or pruning have the potential to shrink the memory footprint of convolutions significantly.

In addition to pruning and quantization, partitioning of models across multiple devices has been applied distributed settings. For example, MoDNN [7]

uses a one-dimensional partitioning scheme where a map-reduce algorithm can execute many of the partitions in parallel. DeepThings [13] uses Fused Tile Partitioning (FTP) to split layers into an even 2D grid and combines them via a fusing process in order for corresponding grid sections to be executed independently. Furthermore, DeepThings proposes data reuse and scheduling approaches such that adjacent partitions can use previously computed data where possible. However, all of these works are designed for computation among several devices. Because of this, communication is a primary consideration. Since MAFAT uses only a single device, alternative techniques such as partial fusing and re-tiling after a certain number of layers can result in more optimal memory usage.

Chapter 2

Background

This chapter explains some of the techniques and background information used in this report. It first discusses how FTP works in DeepThings [13], followed by some background on YOLOv2 [9].

2.1 Fused Tile Partitioning

Fused tile partitioning allows a set of convolutional layers to be split into multiple smaller sub-convolutions. In a given sub-convolution, there are one or more sub-layers. Each layer has input and output data referred to as the input and output tile, respectively. The output tile of layer l is equal to the input tile of layer $l+1$. In the sub-convolutions across layers, multiple tiles can be combined and fused to execute as one unit. This unit is referred to as a task, and can be executed independently of all other tasks. All tasks must have the same sub convolutions and tiles fused in order to have a consistent input and output. Each task also needs to know all of the filter weights for each fused layer.

2.1.1 Tiling Convolutional Layers

A convolution operation is shown in the top portion of 2.1. For a given layer with D_0 channels, there are a D_1 number of filters of some size $F_1 \times F_1$. Each filter has D_0 arrays that correspond to one channel in the input. The filter is then scanned across the image, moving sideways. These filters are shown a the top right of the figure. Each element of the filter has a trained weight to it. The discrete convolution is performed as the multiplication of the input with the weights, and then the sum of all values. The corresponding output is one point in the channel of the output. This operation is sometimes referred to as a dot product. Since the filter relies on input data directly adjacent to current coordinates for a computation, there is no need for input data that is not close to the current region of the input to be present in memory. This allows for an input to be split up, such that only portions of the input are needed in memory at any given time.

The specific method and pattern of splitting a layer into sub-layers is known as tiling. Tiling allows for partial execution and partial loading of input data for a given layer. Tiling also splits convolutional layers along height and width dimensions of the input. Splitting the layer in this way allows for a similar split in the next layer, which means the next tile can be executed immediately after (provided sufficient overlap is considered as discussed in Section 2.1.2). In tiling, the smaller tiled computations are mathematically equivalent to the original, and can be combined into an identical output.

In Figure 2.1, layer 4 is subdivided into an even 3×3 grid. The figure

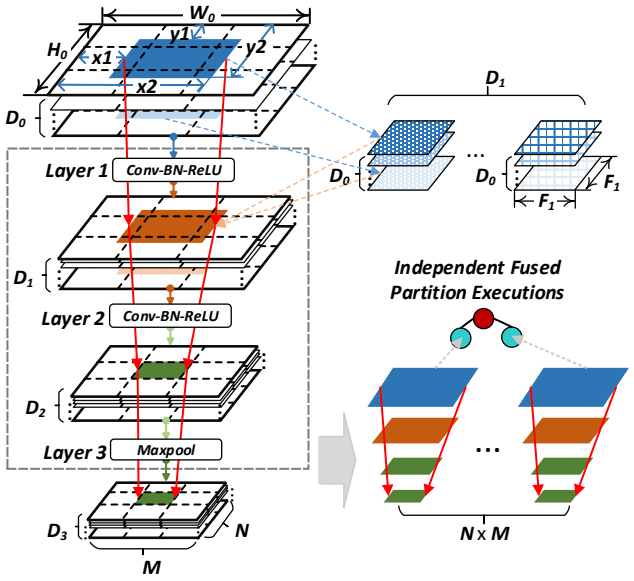

Figure 2.1: An overview of the Fused Tile Partitioning method in [13].

also shows that each tile can be executed independently from the other tiles, and therefore could be executed in any order. There is a small amount of additional overhead for the parameters and other functions. Therefore, it is usually quickest on a single device to use the least number of tiles possible.

2.1.2 Fusing

Fusing is the process of combining similar tiles into one task such that they can be executed independently from other tiles. This fusing task requires all necessary layer weights, which in the case of convolution is all filter weights for each layer. To run, it also needs the input tile to the first layer. Fusing in this manner is ideal for feature-heavy layers as the weights are relatively small

compared to the feature being used.

Critically, since convolutional filters are generally larger than 1×1, each convolution for a point requires the points surrounding it to be added to the convolution. Around the outer edge of the image, this can be addressed with zero-padding. However, on the edge of a tile that is adjacent to other tiles, boundary information must be present to compute the correct output. Therefore, we must pad the tiles with the feature information from an adjacent tile. This produces overlapping data.

This phenomenon is shown in Figure 2.1. The tile size in layer 1 of Figure 2.1 exceeds the bounds of the original grid shape. The larger the number of layers fused, the more information that must be padded to the tile. This overlap with other grids introduces redundant computation, and can add latency.

2.1.3 Data Reuse

Data reuse mitigates the redundant computation introduced by fusing. It shares the overlapped data between adjacent tiles so that one tile is freed of that computation. Figure 2.2 depicts tile A executing first, and then tile B is free to use that information without computing it. This allows high levels of fusing to have comparable computational complexity to the original non-fused and non-tiled variants.

Importantly, to maximize data reuse, it is necessary to develop an ordering of the tiles. The approach taken by [13] is simply to execute every other

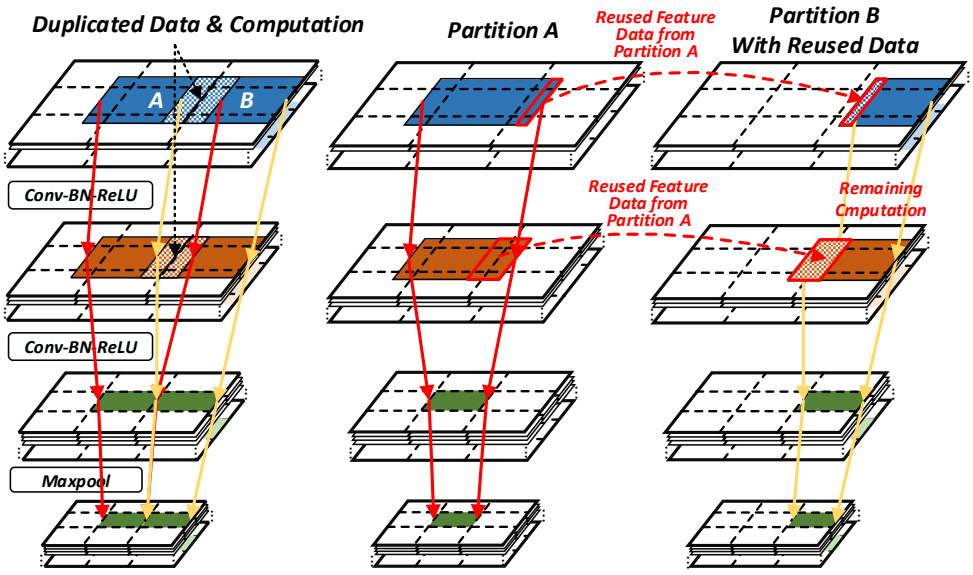

Figure 2.2: An explanation data reuse in [13]

tile (for example 0, 2, 4, etc. when numbered from left to right) to maximize the overlapped data that the other tiles (such as 1, 3) have access to.

2.2 YOLOv2

YOLOv2 is an object detection framework presented in [9]. Its convolutional layers are representative of many modern CNNs. It uses the Darknet network and inference engine. Statistics about the first 16 layers of this model are shown in Table 2.1. This report uses the first 16 layers of the network only to emphasize the feature-heavy optimization. Using MAFAT configurations on weight-heavy later layers will not have any added benefit. For this reason, a single partition or other methods should be considered if these layers exceed

Table 2.1: Data and sizes for the first 16 layers of Darknet.

Layer	Type	Dimensions	Weights	Input	Output	Scratch	Total
0	Conv	608x608x3	3456	4.23	45.13	38.07	87.43
1	Max	608x608x32	0	45.13	11.28	0.00	56.41
2	Conv	304X304x32	73728	11.28	22.56	101.53	135.45
3	Max	304x304x64	0	22.56	5.64	0.00	28.20
4	Conv	152x152x64	294912	5.64	11.28	50.77	67.97
5	Conv	152x152x128	32768	11.28	5.64	11.28	28.23
6	Conv	152x152x64	294912	5.64	11.28	50.77	67.97
7	Max	152x152x128	0	11.28	2.82	0.00	14.10
8	Conv	76x76x128	1179648	2.82	5.64	25.38	34.97
9	Conv	76x76x256	131072	5.64	2.82	5.64	14.23
10	Conv	76x76x128	1179648	2.82	5.64	25.38	34.97
11	Max	76x76x256	0	5.64	1.41	0.00	7.05
12	Conv	38x38x256	4717872	1.41	2.82	12.69	21.42
13	Conv	38x38x512	524288	2.82	1.41	2.82	7.55
14	Conv	38x38x256	4718592	1.41	2.82	12.69	21.42
15	Conv	38x38x512	524288	2.82	1.41	2.82	7.55

Note: The sizes are in Megabytes except for the Weights which are in bytes

memory requirements, which is out of the scope of this report.

The table displays the dimensions, sizes of the weights (the filter parameters), input size, output size, and the scratch memory. The scratch space size of a given layer is the memory that Darknet allocates in order to do a layer calculation. Given output width w, height h, filter length f, stride s and number of channels c, the scratch size is defined for layer l as:

$$scratch = \frac{w \times h \times (f)^2 \times c}{s} \tag{2.1}$$

Clearly, the largest combined memory for a given layer is layer 2. If

that layer is loaded in its entirety, Table 2.1 states that the processor needs at least 135 MB of memory for YOLOv2 to run cleanly. This is a potential explanation for the increase in the time of Figure 1.1 in Section 1.1, even though the input, weights and output only add up to 56 MB.

Chapter 3

Memory-Aware Fusing and Tiling

This report proposes memory-aware fusing and tiling (MAFAT). Instead of fusing all layers to minimize communication, MAFAT separates layers into up to two layer groups to provide additional control over memory usage. The advantage to multiple layer groups is greater tiling flexibility and smaller overlaps. For example, if the early layers take up significantly more data than the later layers, it may make sense to tile the earlier layers more heavily. In this case, there is less memory being used in the earlier layers, but there's no significant added overhead in later layers from unnecessary tiling. Additionally, for a smaller number of fused layers, the overlap incurred will be less. This means that there is less redundant computation, and that the grid of the earlier layers does not have large task size disparities. In a standard 3×3 fused tiling with data reuse, the middle task does not reuse any data. Because of this, it is much larger than the surrounding tiles. This means that the memory usage is disproportionately large for that task. The less overlap, the less this phenomenon occurs, the more even the tasks are, and the less maximum memory used.

3.1 Methodology

MAFAT currently takes any set of n convolutional and maxpool layers (layers which take the maximum value from a local set, and can also be tiled). The layers are configured in a single layer group with all layers fused or two layer groups separated by a $cut < n$. This cut is the point at which the two layer groups are split. The first layer group will be from layer 0 to layer $cut-1$, and the second will be from layer cut to layer n. In this way, each layer is part of one of the two layer groups. Of course, there is some additional overhead. Not only must additional parameters be stored, but the cut layer must be merged in memory and re-tiled.

Potential cuts are determined in a memory-aware fashion. Collecting all the tiled data into a single input tensor and re-tiling can be memory intensive. To make this as efficient as possible, the cuts were chosen to be directly after maxpool layers. After these layers, the tensors are signifiantly smaller, as they have effectively just been down-sampled. In the YOLOv2 example in Table 2.1, the potential cuts are at layers 2, 4, 8, and 12.

For the two layer groups, the tiling for each group is independent of the other. This means that the first layer group could be tiled at 5×5 while the second could be tiled at 2×2. The potential tilings were all even on height and width, and were 1×1, 2×2, 3×3, 4×4, and 5×5.

3.2 Predicting Maximum Memory Usage

We also developed a predictor of the maximum memory usage of a given MAFAT configuration based on the maximum memory usage of the largest tile in each layer group. It was observed that layer groups would generally exceed memory towards the beginning and middle of their execution. Likely, the tasks were swapping due to the large memory requirements of the early layers. Additionally, it was found that the factors that best predicted maximum memory usage were the largest combination of:

- scratch space of tile t
- input to tile t
- output of tile t
- output of previous layer to tile t

While other parameters such as the size of data reuse and size of tasks waiting in the processing queue were considered, these were found to negatively affect the ability of the predictor to accurately predict memory usage. Additionally, weights for all layers in the fusing are assumed to be in memory constantly, as well as a significant amount of additional overhead devoted to network parameters, system variables, and other data. A constant *bias* term of 31 MB was empirically determined to account for these. The bias used is expected to vary should the operating system, network or hardware change.

Since there are two layer groups, both must be checked for the maximum memory that they require. Therefore, each tile in each group needs to be checked in order to calculate the memory usage. The memory predictor is a variation on the Fused Tile Partitioning Algorithm from [13], and is shown in Algorithm 1. This algorithm predicts the maximum memory usage of a given layer group and tiling strategy. The inputs to Algorithm 1 are the parameters of a layer group spanning from layer *top* to layer *bottom* with an $N \times M$ tiling strategy, as well as a network configuration $\mathbf{W}, \mathbf{H}, \mathbf{F}, \mathbf{S}$ with each layer l having width W_l and height H_l, filters of size F_l and a stride of S_l. The stride is how much the filter moves each computation.

The *Grid* function used in Algorithm 1 constructs an even tile grid with no overlap, and assigns the appropriately indexed tile the coordinates and dimensions. The *upTile* function calculates the dimensions of the previous tile by calculating the additional overlap that is required due to the filter and the stride used. The *upTile* function is outlined in [13] as the traversal function.

To cover both of the layer groups that are possible in the network, Algorithm 2 runs the layer group prediction for both cuts and gets the maximum of the two. The maximum of both runs is the maximum of the system. The additional inputs required are a first layer group of size $N_1 \times M_1$ fused until $cut - 1$, and a second layer group of size $N_2 \times M_2$ fused from *cut* until the end of the layers.

Figure 3.1 depicts the predicted memory limit and the measured limit for a single layer group. The measured limit was determined using the setup

Algorithm 1: Memory predictor for a single layer group

predictLayerGroup($N, M, \mathbf{W}, \mathbf{H}, \mathbf{F}, \mathbf{S}, top, bottom$)
$max \leftarrow 0$;
for $i \in 0..N$ **do**
  for $j \in 0..M$ **do**
    $l \leftarrow bottom$;
    $tile_l \leftarrow \text{Grid}(N, M, W_l, H_l, i, j)$;
    while $l \leq top$ **do**
      $w_{in}, h_{in}, w_{out}, h_{out}, c_{in}, c_{out} \leftarrow tile_l$;
      $scratch \leftarrow w_{out} \times h_{out} \times c_{in} \times (F_l)^2 / S_l$;
      $input \leftarrow w_{in} \times h_{in} \times c_{in}$;
      $output \leftarrow w_{out} \times h_{out} \times c_{out}$;
      $mem \leftarrow scratch + output + (input \times 2)$;
      if $mem > Max$ **then**
        $Max \leftarrow mem$;
      if $l < top$ **then**
        $tile_{l-1} \leftarrow \text{upTile}(tile_l, N, M, W_{l-1}, H_{l-1}, F_l, S_l)$;
      $l \leftarrow l + 1$;
return $Max + bias$;

Algorithm 2: Memory predictor for the entire network

predictMem($N_1, M_1, N_2, M_2, \mathbf{W}, \mathbf{H}, \mathbf{F}, \mathbf{S}, cut, n$)
$top \leftarrow 0$;
$bottom \leftarrow cut - 1$;
$firstMem \leftarrow \text{predictLayerGroup}(N_1, M_1, \mathbf{W}, \mathbf{H}, \mathbf{F}, \mathbf{S}, top, bottom)$;
$top \leftarrow cut$;
$bottom \leftarrow n$;
$secMem \leftarrow \text{predictLayerGroup}(N_2, M_2, \mathbf{W}, \mathbf{H}, \mathbf{F}, \mathbf{S}, top, bottom)$;
return $\text{Max}(firstMem, secMem)$;

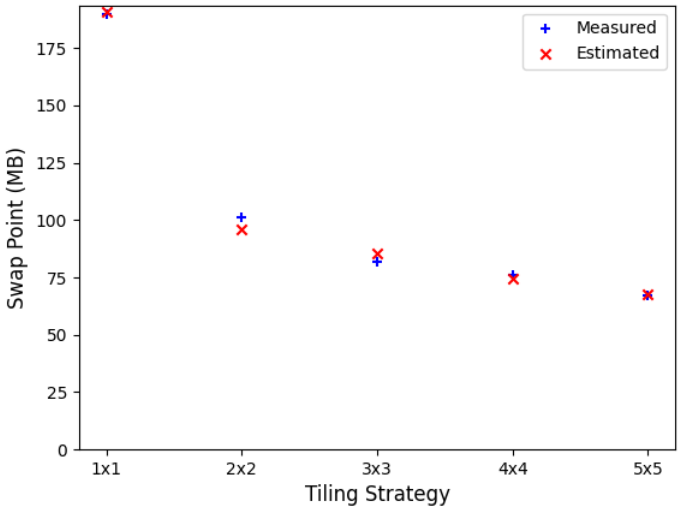

Figure 3.1: Predicting the memory usage for fully fused 16 layers.

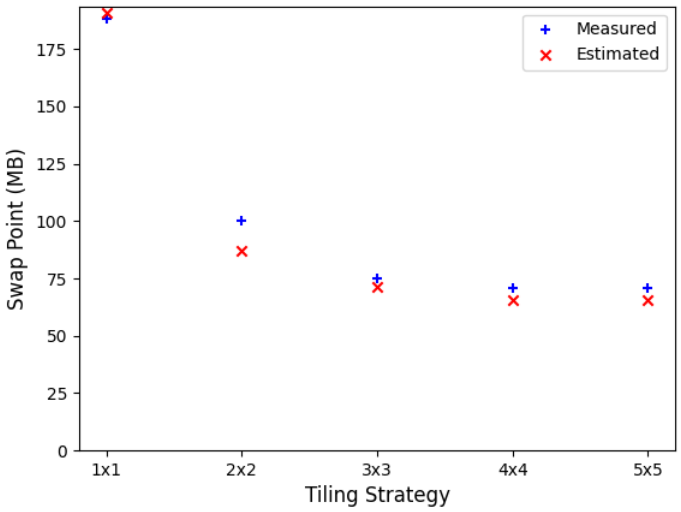

Figure 3.2: Predicting memory usage for fused 8 layers with a 2x2 fused tiling on layers 9-16.

in Section 4.2 by decreasing the memory constraint in 1 Megabyte increments until swaps were observed. Figure 3.2 shows the predicted values against the measured values for the MAFAT configurations with a cut at layer 8 and a 2×2 bottom tiling strategy. The predictor still performs well.

3.3 Configuration Algorithm

To determine the ideal MAFAT configuration, Algorithm 3 performs a search over a subset of the configuration space. Its goal is to return a near-optimal configuration of the network such that the end latency will be as small as possible. The inputs to the algorithm are the layer parameters and the memory limit. The relevant layer parameters are width (W_l), height (H_l), filter size (F_l), stride (S_l), and the total number of layers to be fused (n). The vector of potential cuts ($Cuts$) is specific to YOLOv2, due to the location of the maxpool layers. The restriction of the search space is based on a manual search of the configuration space and best results in that space. Specifically, no latency advantage was found for cuts at layer 4, and when there were cuts made, the best performing second layer group tiling was 2×2. Therefore, the search can be simplified dramatically. The tiling strategies are also currently limited to even squares. The final restriction to the space is present on lines 9 and 10. It was observed that cuts at layer 12 and later with a large number of tiles developed more overlapped data and overhead than smaller cuts, and are never optimal. Thus, they are excluded from the search space.

Algorithm 3 returns the number of tiles for the first layer group LG_1,

Algorithm 3: Configuration search algorithm

```
1  getConfig(W, H, F, S, n, MemoryLimit)
2  Cuts ← {16, 12, 8};
3  Tiles ← {1, 2, 3, 4, 5};
4  LG_2 ← 4;
5  N_2 ← LG_2;  M_2 ← LG_2;
6  l ← getMaxLayer(NetworkParams);
7  for cut ∈ Cuts do
8      for tile ∈ Tiles do
9          LG_1 ← tile;
10         N_1 ← LG_1;  M_1 ← LG_1;
11         if cut ≥ 12 and tile > 2 then
12             continue;
13         else if predictMem(N_1, M_1, N_2, M_2, W, H, F, S, cut, n) <
               MemoryLimit then
14             return LG_1, LG_2, cut;
15 return LG_1, LG_2, cut
```

the cut *cut*, and the tiling for the second layer group LG_2. It performs the modified search starting at the highest memory value, and slowly creates more even configurations that require more overhead, but fit in smaller memory footprints. If a configuration is found that fits in the memory limit, there is no unexplored configuration in the search space that will produce a higher memory prediction. Therefore, the latency returned should be the lowest. If virtual memory is enabled, this algorithm assumes that any additional swaps from the operating system will be slower than picking a better configuration. If no configuration can be found, then the algorithm returns the most even configuration: 5×5 into 2×2 with a cut at layer 8.

The intuition behind the algorithm is based on additional tilings creating additional redundancy. Therefore, the algorithm greedily attempts to find the fewest tiles it can use.

Chapter 4

Experimental Results

We applied MAFAT to the YOLOv2 object detection network. The measurements were all carried out on a Raspberry Pi3 running Raspian. The Raspberry Pi was equipped with a quad-core 1.2GHz ARM Cortex-A53 processor and a total memory size of 1 GB. During the measurements, we restricted the Raspberry Pi to a single core and a variable amount of memory from 16MB up to 256MB.

This chapter describes the measurement process used, the manual exploration of the configuration space, as well as the results of automatically optimized MAFAT against the standard Darknet implementation.

4.1 Measuring Swapping, Memory Usage, and Latency

A separate measurement thread was created to measure system swaps in and out of memory each second. This gives information about likely places for a bottleneck. This was achieved using the `vmstat` command. Due to the `vmstat` only working at a full system level, it was crucial to keep the test environment free of as many other running processes as possible. Despite this, there is some noise in the swap measurement.

To measure memory usage of just the process, an additional thread was used that polled the process using the `ps` command. This way we could filter out other processes without as much added system noise. This was useful in seeing more accurately where the swapping would line up with the program.

Both of these threads however added some additional memory usage and could potentially increase swaps or create conflicts with the process. Therefore, when the latency for the process was calculated, internal measurements were used via the `chrono.h` library in C++ for accurate, wall clock times at a millisecond granularity. This also allowed for precise measurements at the beginning and end of an inference. In this report, the latency was measured before the input image was loaded and after the first 16 layers had executed.

4.2 Constricting Memory and CPU Size

To mimic a smaller edge device with minimal effort, this report used control groups. Specifically, the `cpuset` and `memory` control groups were used to restrict the experiment to a single core and a smaller amount of memory, respectively. This allowed for finer adjustments of memory constraints without the need for rebooting. For predictability and reproducibility, as few active processes as possible were running during final latency measurements.

4.3 Manual Exploration

To develop the algorithm, and to better understand the configuration performance, we first performed a manual search of different possible configurations. In the following, a MAFAT configuration with a top layer group tiling of $N_1 \times M_1$, a cut a layer c, and a bottom layer group tiling of $N_2 \times M_2$ is written as $N_1 \times M_1/c/N_2 \times M_2$

Using prior knowledge, the search space of possible cuts was restricted. As mentioned in Section 3.1, intermediate data is reduced the most by cutting the network into two layer groups at layers 4, 8, and 12, or no cut at all. In each case, all layers up to and after the cut were fused together. Additionally, the final layers were split into either 2×2 or 3×3 tiles for reducing maximum memory while still allowing for faster processing times. The tilings for the top layer group were swept from 1×1 to 5×5.

Figure 4.1 shows the effect of top layer group tiling strategies on measured latency across a shrinking memory limit. Each line represents the tiling of the top layer group, which is then cut at layer 8 and fed into a 2×2 bottom layer group. Figure 4.1 demonstrates the superiority of finer tilings in smaller memory footprints, but also the additional overhead they generate when more memory is available. For high memory values in excess of 200 MB, the 1x1 tiling is best. On the other hand, using a 4×4 or 5×5 tiling scheme yields much better results for lower memory values.

Figure 4.2 shows the effect of cut placement and bottom layer group

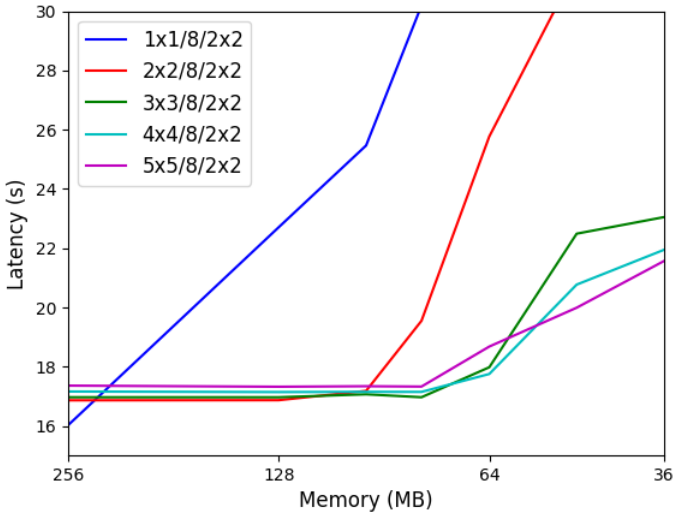

Figure 4.1: Latency for different tilings on a cut at layer 8.

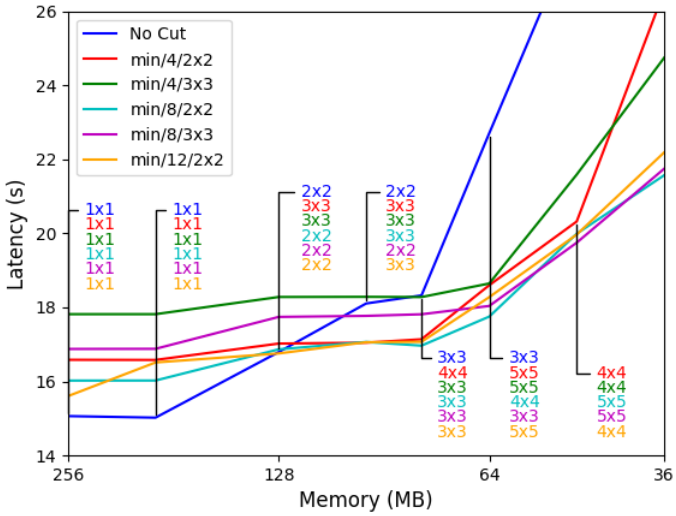

Figure 4.2: Latency for different cut configurations.

tiling strategy on latency for varying memory limits. The top tiling for this line is the tiling strategy (from 1×1 to 5×5) that yields the smallest latency for the given cut and bottom tiling. The best top tiling for each configuration is also annotated onto the graph at each memory point. For example, the $min/8/3 \times 3$ line represents a cut at layer 8 with the best top tiling and a 3×3 bottom tiling. It can therefore be viewed as the optimized top tiling for a given cut and bottom tiling. As seen in the graph, middle cuts at layer 8 have the fastest latency at tighter memory restrictions. It is also clear that the absence of a cut becomes costly at tighter restrictions due to additional layer overlapping. This figure also reinforces Figure 4.1 to show that finer tiling perform better at tighter memory restrictions.

Figure 4.3 compares the best measured latency obtained by the MAFAT manual exploration and search algorithm latencies to the original latencies measured from the standard Darknet Implementation across decreasing memory limits. It is clear from the figure that MAFAT outperforms Darknet and reduces the latency and swaps of an inference.

Interestingly, the minimum configuration for the algorithm, $5 \times 5/8/2 \times 2$, is predicted to have a maximum memory usage of 66 MB. Currently, therefore, there is not a MAFAT configuration that does not run in less than a 66 MB footprint without swapping. However, as memory restrictions get even tighter, the latency increases at a much slower rate than Darknet. This shows that the MAFAT configuration also performs much better under swapping due to more even memory usage across the execution of the network.

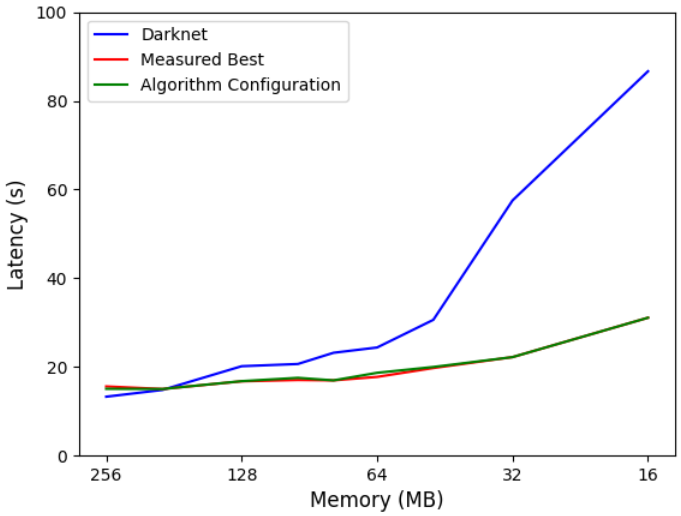

Figure 4.3: Darknet latency compared to algorithm and minimum latency measured.

4.4 Algorithm Performance

Figure 4.3 also plots the measured performance for the configurations produced by our optimization algorithm. The differences between the algorithm and the best measured are shown to be minimal. The algorithm's specific configuration compared to the best measured can be found in Table 4.1. To evaluate algorithm performance, the outputs of the algorithm were calculated for the memory values in the table. This allowed for easy comparison with the existing measured data. Notably, the latency values are quite similar and are all within 6 percent of the best measured from manual exploration. Given how the algorithm relies on prior knowledge and some of the data already recorded, this level of performance is not surprising. However, the intuition behind the

Table 4.1: Comparison of configurations and latencies.

	Best Measured		Algorithm	
MB	Configuration	Latency (ms)	Configuration	Latency (ms)
256	1x1/NoCut	15065	1x1/NoCut	15065
192	1x1/NoCut	15023	1x1/NoCut	15023
128	2x2/12/2x2	16757	2x2/NoCut	16795
96	3x3/4/2x2	17048	2x2/12/2x2	17543
80	3x3/8/2x2	16968	3x3/8/2x2	16968
64	4x4/8/2x2	17753	5x5/8/2x2	18679
48	5x5/8/3x3	19749	5x5/8/2x2	19991
32	5x5/8/2x2	22215	5x5/8/2x2	22215
16	5x5/8/2x2	31095	5x5/8/2x2	31095

algorithm and the basic results should help apply it in other domains.

Chapter 5

Summary and Conclusions

This report presents memory-aware fusing and tiling (MAFAT), an expansion of existing fusing and tiling strategies in order to make feature-heavy convolutional neural network layers feasible on smaller edge devices. Originally, edge devices would have increasing latency measurements due to swapping data between the memory and disk. Many edge devices cannot spare 200 MB to run early convolutional layers, so we break up each layer into sub-convolutions that can then be grouped together and executed in a much smaller memory footprint. This report shows that certain configurations of tiling can offer a respectable 1.37 speedup compared to the naive approach at 64 MB and up to a 2.78 speedup with only 16 MB available.

Additionally, the intuition and structure behind the memory usage of the process is explored, and a simple algorithm is proposed to predict the maximum memory usage of a MAFAT configuration. Given this, an appropriate configuration can be returned for a user to use that is within 6 percent of the best measured latency from a manual exploration.

The code used to take these measurements can be found at [2]. This research area can be further improved by use variable tiling, where each end

tile is not the same size. We believe this could allow for reduced task size variation, and thus smaller footprints. We also want to explore this algorithm and see how well the predictor applies to other CNNs on the edge. While the intuition is useful and true, calculating the bias and the choice of variables could restrict the algorithms' overall efficacy. Currently, the end user must get a feel for possible different measurements and what cuts make sense. Additionally, more than two tiles and even larger tiling strategies, such as 6×6 could be useful, especially in super-low memory constraints. Finally, more sophisticated algorithms could be used to predict amounts of swapping as well and make more optimal and exhaustive recommendations.

Acknowledgments

This research was also sponsored by the Army Research Office and was accomplished under Cooperative Agreement Number W911NF-19-2-0333. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Office or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

References

[1] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Structured pruning of deep convolutional neural networks. *Journal of Emerging Technologies in Computing Systems*, 13(3), 2017.

[2] Jackson Farley. MAFAT. `https://github.com/JacksonFarley/MAFAT`, 2021. Online.

[3] Yunchao Gong, L. Liu, Ming Yang, and Lubomir D. Bourdev. Compressing deep convolutional networks using vector quantization. *ArXiv*, abs/1412.6115, 2014.

[4] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *ArXiv preprint ArXiv:1807.05358*, 2018.

[5] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. In *5th International Conference on Learning Representations, ICLR*, 2017.

[6] Darryl Dexu Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In *33nd International Conference on Machine Learning, ICML*, 2016.

[7] Jiachen Mao, Xiang Chen, Kent W. Nixon, Christopher Krieger, and Yiran Chen. MoDNN: Local distributed mobile computing system for deep neural network. In *Design, Automation Test in Europe Conference Exhibition, DATE*, 2017.

[8] Jihong Park, Sumudu Samarakoon, Mehdi Bennis, and Mérouane Debbah. Wireless network intelligence at the edge. *Proceedings of the IEEE*, 107(11):2204–2239, 2019.

[9] Joseph Redmon and Ali Farhadi. Yolo9000: Better, faster, stronger, 2016.

[10] B. Verhoef, Nathan Laubeuf, S. Cosemans, P. Debacker, I. Papistas, A. Mallik, and D. Verkest. FQ-Conv: Fully quantized convolution for efficient and accurate inference. *ArXiv*, abs/1912.09356, 2019.

[11] Kai Yang, Tao Jiang, Yuanming Shi, and Zhi Ding. Federated learning via over-the-air computation. *IEEE Transactions on Wireless Communications*, 19(3):2022–2035, 2020.

[12] Kai Yang, Yuanming Shi, Wei Yu, and Zhi Ding. Energy-efficient processing and robust wireless cooperative transmission for edge inference. *IEEE Internet of Things Journal*, 7(10):9456–9470, 2020.

[13] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. DeepThings: Distributed adaptive deep learning inference on resource-

constrained IoT edge clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2348–2359, 2018.

[14] Li Zhou, Mohammad Hossein Samavatian, Anys Bacha, Saikat Majumdar, and Radu Teodorescu. Adaptive parallel execution of deep neural networks on heterogeneous edge devices. In *4th ACM/IEEE Symposium on Edge Computing, SEC*, 2019.