This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2018.2857278, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

Auto-Tuning CNNs for Coarse-Grained Reconfigurable Array-based Accelerators

Inpyo Bae Barend Harris Hyemi Min Bernhard Egger
Department of Computer Science and Engineering
Seoul National University, Seoul, Korea

***Abstract*—As more and more deep learning tasks are pushed to mobile devices, accelerators for running these networks efficiently gain in importance. We show a that an existing class of general purpose accelerators, modulo-scheduled coarse-grained reconfigurable array (CGRA) processors typically used to accelerate multimedia workloads, can be a viable alternative to dedicated DNN processing hardware. To this end, an auto-tuning compiler is presented that maps convolutional neural networks (CNNs) efficiently on such architectures. The auto-tuner analyzes the structure of the CNN and the features of the CGRA, then explores the large optimization space to generate code that allows for an efficient mapping of the network. Evaluated with various CNNs, the auto-tuned code achieves an 11-fold speedup over the initial mapping. Comparing the energy per interference, the CGRA outperforms other general-purpose accelerators and an ARMv8 processor by a significant margin.**

***Index Terms*—CNN accelerator, coarse-grained reconfigurable array, automatic code optimization**

I. Introduction

In recent years, Deep Neural Networks (DNNs) have seen an unprecedented performance improvement in tasks such as computer vision and speech recognition. Today, DNNs outperform humans in terms of not only speed but also accuracy in various complicated tasks [1], [2]. For larger systems, GPUs and TPUs [3] can provide effective acceleration, however, when deploying DNNs in more diverse areas such as IoT systems, medical computing or automated vehicles, acceleration on less powerful embedded devices gains in importance. Many specialized hardware DNN accelerators for such environments have recently been proposed [4], [5], [6], [7], [8].

At the same time, the trend of high-quality audio and high-resolution displays or camera sensors on embedded devices has led to a greatly increased demand for solutions that are able to process a high computational load at minimal power consumption. Typical choices for such application domains are FPGAs, ASICs, or Coarse-Grained Reconfigurable Arrays (CGRAs). CGRAs provide the necessary computational power at a moderate energy consumption while remaining easily programmable [9], [10] and have been integrated into commercially-available embedded systems such as cameras, smartphones, tablets, or smart TVs [11], [12], [13], [14].

An analysis of recently proposed hardware accelerators for DNNs on embedded systems reveals that the presented designs share many similarities with CGRA architectures. Both are composed of a spatial array of processing elements (PEs) with data flowing through an interconnection network. Memory is located as close as possible to the computational units to accommodate the high memory bandwidth demands of CNNs and multimedia workloads. Given that CGRAs are already integrated into the MPSoC of embedded devices to accelerate multimedia workloads [15], a natural question is then whether such accelerators could be used, maybe with a few architectural modifications, to also efficiently execute neural networks. This would enable the use of the CGRA for both its existing applications as well as CNN acceleration, reducing the need for more dedicated hardware.

Executing CNNs efficiently on modulo-scheduled CGRAs, however, is a complex task. CNNs consist of several layers of varying sizes and data types. Each layer consist of several nested loops with different properties. To run these loops efficiently both the loops' properties and the CGRA architecture must be considered. Ideal loop optimizations for each loop are not easily attainable as their effect on the resulting number of CGRA cycles is complex.

To address these issues, this paper proposes a novel auto-tuning framework for modulo-scheduled CGRAs. The framework analyzes both the structure of the CNN and the architecture of the CGRA to then automatically select the best algorithm and explore the design space of loop optimizations for each individual layer of the neural network.

An evaluation with several CNNs shows that the auto-tuner is able to improve CGRA performance, on average, by a factor of 11 compared to a mapping of the unoptimized code. Compared to an ARMv8 processor executing the same code, an over 50x improvement is achieved; and even with a dedicated CNN ARM library, the CGRA is still over 5 times faster than the 4-core ARMv8 processor. In terms of energy consumption per inference, the auto-tuned code outperforms other general-purpose processors and accelerators by a significant margin, demonstrating the potential of CGRAs as a viable alternative for on-device inference of CNNs.

II. Background

A. Coarse-Grained Reconfigurable Arrays

CGRA accelerators have been proposed as an alternative to FPGAs and ASICS. While the former offer bit-level flexibility at the expense of a long reconfiguration time, the latter achieve high energy efficiency but suffer from high development cost

This article was presented in the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES) 2018 and appears as part of the ESWEEK-TCAD special issue.
Corresponding author: Bernhard Egger (bernhard@csap.snu.ac.kr)

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2018.2857278, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

and inflexibility. CGRAs are positioned in between, offering programmability and fast reconfiguration at a moderate energy consumption. The components of a CGRA are an array of typically heterogeneous processing elements (PEs), global and local register files, constant units, and an often irregular interconnection network that connects the different hardware units. CGRAs are reconfigurable at the word-level and the hardware units are programmable in each execution cycle.

This work focuses on a class of hybrid VLIW/CGRA processors based on the ADRES architecture [9] such as the Samsung Reconfigurable Processor (SRP) [16]. Different to pure accelerators, the hybrid architecture allows execution of control-flow-intensive, general-purpose code as well as software-pipelined data-flow code on the same processor, eliminating costly control transfers between a host processor and the accelerator. Innermost loops are mapped to the CGRA accelerator using a a technique called modulo-scheduling (Section V-C1). This class of CGRAs supports predicated execution, allowing the mapping of loops that contain control-flow instructions [17]. The use of the SRP to evaluate our automatic optimization techniques is motivated by the fact that the processor has been integrated into commercial MPSoCs for smartphones, TVs, printers, and cameras [11], [12], [13], [14].

B. Convolutional Neural Networks

DNNs consist of one input layer, several hidden layers, and one output layer. The inputs to neuron k in layer l are the products of the output values of every connected neuron x_i in layer $l-1$ with trainable weights assigned to the connections w_{ki}. The output value y_k of the neuron is computed by an activation function φ that takes the sum of these products plus a trainable bias b_k as its input.

$$y_k = \varphi\left(\sum_{i=0}^{m} w_{ki}x_i + b_k\right) \tag{1}$$

Convolutional Neural Networks (CNNs) are a subset of DNNs that use convolutional layers where a small kernel of weights is moved across the input to compute a convolution. In terms of implementation, the outputs of a layer are generated by matrix multiplications, with several algorithms such as naïve convolutions, GEMM [18], or Winograd [19] having been proposed to compute the output values of a convolutional layer.

C. Processing CNNs with CGRAs

CGRAs bear a lot of resemblance to dedicated DNN processors, the key similarity being the use of spatial architectures consisting of multiple PEs with memory situated close by. These architectures are well suited for CNN processing because the bulk of operations concerns memory operations to access the inputs and weights and floating point operations to perform matrix multiplications. Inputs and weights are stored in on-chip memory to reduce the number of off-chip accesses. Registers can further be used to store shared weights and input values even closer to the PEs. The multiple PEs allow the parallelism of the calculations to be exploited.

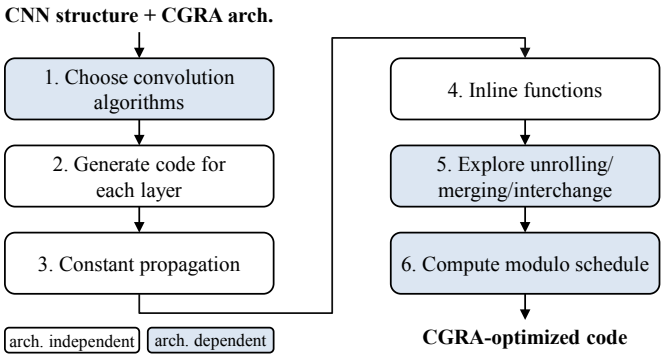


Fig. 1: Optimization Steps.

Convolutional and fully-connected layers consist of nested loops with a high iteration count and simple loop bodies that load and multiply the input values from layer $l-1$ with the weights of layer l to produce the output of layer l. This structure allows for CGRA-specific optimizations.

(i) the loop-level parallelism (LLP) of the innermost loop body can be exploited using software pipelining
(ii) the instruction-level parallelism (ILP) can be increased using loop unrolling
(iii) the limited number of loop-carried dependencies allows the interchange of loops to further increase the ILP

While hand optimization by an experienced engineer is possible, the number of differing loops in the CNN and the large parameter space renders manual optimization in reasonable time an infeasible task, calling for an automated method. The following sections describe such a framework.

III. Auto-Tuning Framework

Key to achieve good performance on a modulo-scheduled CGRA is to have an innermost loop with a large number of iterations, sufficient ILP, and no function calls. The auto-tuner framework performs the code analysis and transformations on the abstract syntax tree (AST) of the code immediately before invoking the modulo scheduler. Figure 1 details the individual steps. Architecture-independent steps are shown with white, steps that depend on the CGRA architecture are shown with a blue-gray background.

Inputs to the framework are the structure of the CNN and a high-level architecture description of the CGRA. Steps 1 and 2 select the convolutional kernel implementations and generate C code for each for each layer of the CNN; this process is further detailed in Section IV. Constants such as the batch size, the height and width of the input and the number, dimension, and stride of the convolutional kernels are known and propagated into the symbolic loop bounds in step 3, then function calls to activation functions such as `ReLU` in innermost loops are inlined (step 4). Employing a runtime cycle-estimation model, step 5 explores the best combination of loop unrolling, loop merging, and loop interchange. This process is described in detail in Section V. Finally, in step 6, the best combination is applied to the AST which is then mapped onto the CGRA by a modulo scheduler [20], [21].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2018.2857278, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

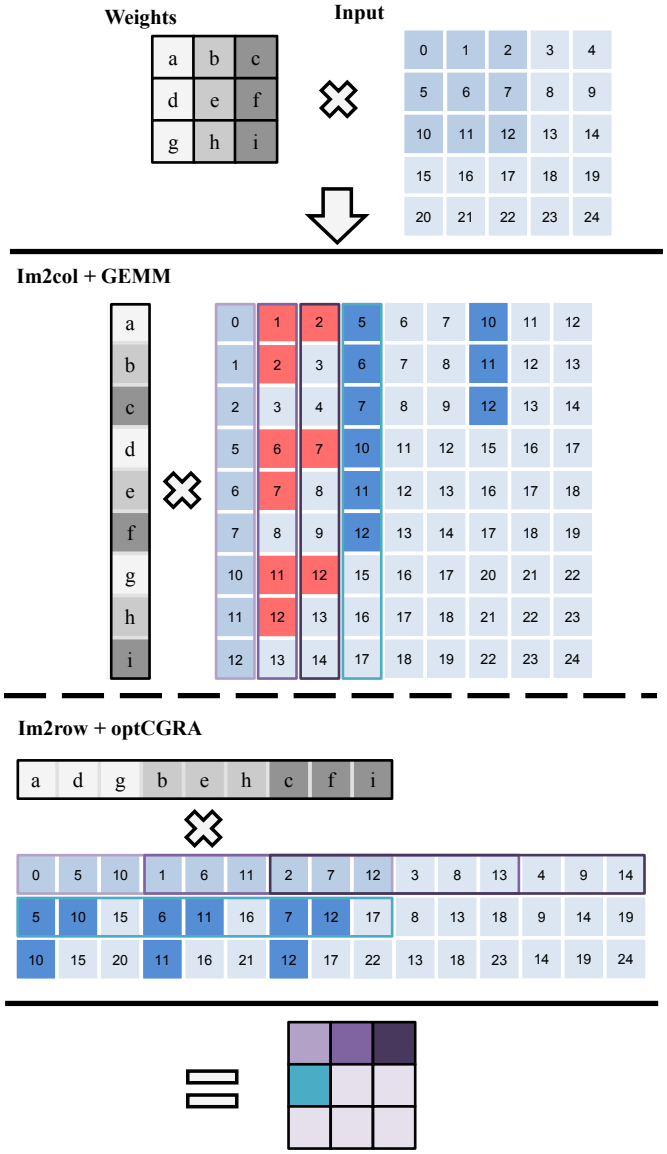


Fig. 2: Computation of a convolution with `im2col+GEMM` and `im2row+optCGRA`.

IV. CGRA-optimized Convolutions

One of the most commonly used algorithms to calculate CNN convolutions is `GEMM` [22]. The multi-dimensional parameter space is first transformed into two 2D matrices in a step called `im2col`, after which the optimized function `GEMM` multiplies the two matrices to produce the desired output. This transformation is also advantageous for CGRAs as it leads to an innermost loop with a high iteration count that can be easily software pipelined. However, if the convolutional window overlaps between iterations (e.g. with a 3×3 kernel and a stride of 1) a large amount of data is duplicated and a significant number values are loaded multiple times.

Figure 2 illustrates a convolution with `im2col + GEMM`. `Im2col` enlarges the original 5x5 input into a 9x9 matrix whose columns can be multiplied in parallel with the weight vector. Duplicated data values due to the horizontal shifting of the convolutional kernel are shown in red, duplication introduced by vertical shifting is colored blue. In the best case, each input value is only loaded once; however, the data duplication of `im2col` causes 27 memory operations to each 3x3 subarray in the input instead of the optimal 9 operations.

We propose a CGRA-tailored convolution algorithm that exploits the large number of registers of CGRAs and reduces the number of memory operations. The 2D input is first transposed into rows by the `im2row` function as shown in Figure 2. This operation duplicates data only in the vertical dimension; again shown in blue. A modified convolution algorithm optimized for CGRA, `optCGRA`, is then used to calculate the result. `OptCGRA` reuses data values in the horizontal dimension: for a $k \times k$ convolutional kernel and each subsequent convolution in the horizontal direction, only a $1 \times k$ column of input values is loaded from memory, while the $k \times k$ kernel weights and the $k \times (k-1)$ input values are kept in registers. According to the taxonomy in [6], `im2row+optGEMM` employs weight stationary, output stationary, and column stationary data flow. Compared to `im2col+GEMM`, the proposed `im2row+optCGRA` reduces the number of redundant memory accesses by 50% from 18 to 9.

`OptCGRA` is also a natural fit to unrolling. Two or more rows can be processed in one iteration, allowing for an increased ILP in the innermost CGRA-mapped loop. The architecture of the CGRA dictates the optimal loop unrolling factor in dependence of the number of PEs and number of local registers. A comparison of the code for `GEMM` and `optCGRA` for a 3×3 kernel is shown in Figure 3.

V. Auto-Tuning for CGRAs

The proposed framework applies a number of architecture-independent and -dependent optimizations before the CGRA modulo scheduler is invoked. The goal of auto-tuning is to transform a given loop nest into one that has an innermost loop with a high iteration count and sufficient ILP.

A. Architecture-Independent Optimizations

1) Layer Separation & Constant Propagation: To perform subsequent architecture-dependent loop optimizations, it is advantageous that the iteration bounds of each loop are known. The framework generates explicit code for each layer of the CNN and known constant parameters such as the number and dimensions of the CNN kernels and the input are propagated into the code. Figure 4 (a)+(b) shows an example of this transformation (note the loop bounds).

2) Function Inlining: Loop bodies containing function calls cannot be software-pipelined [23], however, the activation function φ (see Equation 1) is typically invoked with a function call. Calls to activation functions into the bodies of the innermost loops are thus inlined to increase the amount of code that can be mapped to the CGRA accelerator.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2018.2857278, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

```
#define I out_channel
#define J width*height
#define K in_channel*kernel*kernel

// A: weight, B: input(transformed) C: output
for (i=0; i<I; i++) {
  for (j=0; j<J; j++) {
    for (k=0; k<K; k++) {
      C[i*J +j] += A[i*K + k] * B[k*J + j];
    }
  }
}
```

(a) GEMM

```
#define I out_channel
#define J in_channel
#define K height
#define L width*kernel
#define AREA width*height

// 3x3 convolution
// A: weight, B: input(transformed) C: output
for (i=0; i<I; i++) {
  for (j=0; j<J; j++) {
    float c = 0;
    // Load 3x3 weight kernel in height orientation
    float a0 = A[(i*J+j)*9 + 0], a1 = A[[(i*J+j)*9 + 3], …, a8 = A[[(i*J+j)*9 + 8];
    for (k=0; k<K; k++) {
      // Load first 3x2 entries of input row
      float b0 = B[(j*K+k)*L + 0], b1 = B[(j*K+k)*L + 1], …, b5 = B[(j*K+k)*L + 5];
      for (l=6; l<L; l+=3) {
        // Stride is 1, so 3 new loads
        float b6 = B[(j*K+k)*L + l+0], b7 = B[(j*K+k)*L + l+1], b8 = B[(j*K+k)*L + l+2];

        // Convolution
        t0 = a0 * b0; t1 = a1 * b1; …; t8 = a8 * b8;

        C[i*AREA + c++] += t0 + t1 + … + t8;

        // Reuse overlapped inputs
        b0 = b3; b1 = b4; b2 = b5; b3 = b6; b4 = b7; b5 = b8;
      }
    }
  }
}
```

(b) optCGRA

Fig. 3: Implementation of `GEMM` and `optCGRA`.

B. CGRA-Specific Optimizations

1) Selection of the Convolution Algorithm: The CGRA-optimized `optCGRA` convolution algorithm stores the $k \times k$ weights of a kernel and $k \times (k-1)$ input values in registers across a loop iteration; even more if the loop body is unrolled. If the number of values held in registers is close to the number of physically available registers in the CGRA, the modulo scheduler is unlikely to find a good mapping of the loop; in such cases, the less register-intensive `GEMM` should be chosen.

The auto-tuner selects the convolution algorithm of a layer as follows: for $k = 1$, i.e., 1×1 convolutions, there is no data duplication and `GEMM` can be used without invoking `im2col`. For larger convolutions, `im2row+optCGRA` is chosen if the number of values held in registers, $2 \times k^2 - k$, is smaller than the total number of data registers in the CGRA divided by 2. Otherwise, `im2col+GEMM` is selected.

2) Loop Interchange and Merging: Loop interchange allows moving an outer loop with a high iteration count to the innermost position if there are no dependencies that prevent such an operation. The framework analyzes loop-carried dependencies and performs only valid loop interchange operations. `OptCGRA` contains loop-carried dependencies, whereas `GEMM` has no such constraints and the loop with the highest number can be moved to the innermost position. When exploring unrolling factors both interchange variants are considered and the one resulting in the lowest estimated cycles is chosen.

In addition to loop interchange, the auto-tuner also applies loop merging to increase the number of loop iterations. For example, small x/y loops that iterate over the two dimensions of an input image can be combined into a larger xy loop, and the values of x and y are computed inside this loop. Note that the targeted CGRA supports predicated execution [17] and is thus able to execute loops containing branches. Loop interchange and merging are illustrated in Figure 4 (b)+(c).

3) Loop Unrolling: Loop unrolling combines the loop bodies of several iterations into one, allowing for a higher ILP. For a loop nest, the best combination of loop unrolling factors (UF) not only depends on the CGRA architecture but also the number of iterations at each level of the loop. Determining the performance of unrolling is not easy for modulo-scheduled loops because of the long compilation time and the complex non-linear interaction of hardware- and data-dependencies on performance (Section V-C1). For a small number of loops, a human CGRA expert is able to determine the ideal unrolling factors in reasonable time. To give some context, manually discovering the best unrolling factors for the loops of a small network consisting of only 1 convolutional, 1 pooling, and 1 fully-connected layer by an experienced CGRA engineer took approximately one work week. Finding the optimal unrolling factors for the many distinct loop nests generated by the proposed framework thus calls for an automated process.

While the these loop optimization techniques are not new, finding a good combination of the available optimizations for a given CGRA instance is difficult. The main obstacle is the long compilation time of modulo schedulers that precludes an automated process that computes the modulo schedules for all possible unrolling factors. We therefore propose a model that is able to predict the performance of an unrolled loop nest. The model is then used to estimate the execution cycles of every unrolling factor combination for each loop; the best combination is chosen and sent to a modulo scheduler for compilation. This process is described in detail in the following Section V-C.

C. Determining the Unrolling Factors

1) Modulo Scheduling: Execution of a loop on a CGRA is guided by a software-pipelined schedule that overlaps different iterations of a loop body in time and executes them in parallel. The Initiation Interval (II) of a schedule determines after how many cycles the next loop iteration is initiated; with an $II = 2$, for example, a new loop iteration is started every 2 cycles. The throughput of a loop approaches II cycles for a large number of iterations; the smaller the II, the better the performance of the loop.

The lower limit of the II, the Minimal Initiation Interval defined as $MII = max(ResMII, RecMII)$, is determined by the available hardware resources (Resource-constrained MII, ResMII) and loop-carried dependencies (Recurrence-constrained MII, RecMII). For a CGRA with 2 PEs, for example, the ResMII for a loop body with 6 operations is $ResMII = 6/2 = 3$. To understand the RecMII, consider a

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2018.2857278, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

```
for (y=0; y<HEIGHT; y+=STRIDE)
  for (x=0; x<WIDTH; x+=STRIDE)
    for (k=0; k<KERNEL; k++)
      for (c=0; c<CHANNEL; c++)
      {
        // loop body
      }
```

(a) template code

```
for (k=0; k<16; k++)
  for (y=0; y<32; y+=1)
    for (x=0; x<32; x+=1) {
        // loop body for c=0
        // loop body for c=1
        // loop body for c=2
    }
```

(b) after unrolling and loop interchange

```
for (k=0; k<16; k++) {
  y = next_y = next_x = 0;
  for (yx=0; yx<32*32; yx+=1) {
    x = next_x; next_x += 1;
    if (next_x >= 32) {
      y=next_y; next_y+=1; next_x=0;
    }
    // loop body for c=0..2
}
```

(c) merging x/y loops and strength reduction

Fig. 4: Loop Transformations

loop in which an instruction executed in cycle 5 of iteration i produces a value that is consumed by an instruction in cycle 2 of loop iteration $i+1$. The RecMII has to be larger or equal to $5-2=3$ cycles to satisfy this dependency.

A modulo schedule is computed in an iterative process starting with $II = MII$. The scheduler tries to map all operations of a loop body onto the available PEs while respecting all dependencies. An operation placed at time t is mapped to a PE at time $(t \bmod II)$, hence the name *modulo scheduling*. The scheduler also has to ensure that input operands are properly routed from their producers to the inputs of a PE at a given time. If no schedule can be found at the current II, the scheduler increases II by one and starts over [23].

2) Finding the Best Unrolling Factors: The loops of a loop nest can be unrolled at every level. Determining the best loop unrolling for a loop nest with nesting depth n thus requires finding a vector containing n unrolling factors, one for each level of the loop. This vector can be found by performing a brute force search through all unrolling factor combinations and calculating the execution cycles of the unrolled loop nest.

Special care has to be taken if the loop iteration count $iter$ is not divisible by the unrolling factor UF. For such cases, the original loop is split into an unrolled loop iterating from 1 to $\lfloor iter/UF \rfloor$ at an increment of UF, and a loop that takes care of the remaining iterations from $\lfloor iter/UF \rfloor$ to UF. This loop can also be unrolled; we recursively explore the unrolling factors for all loops as illustrated Figure 5.

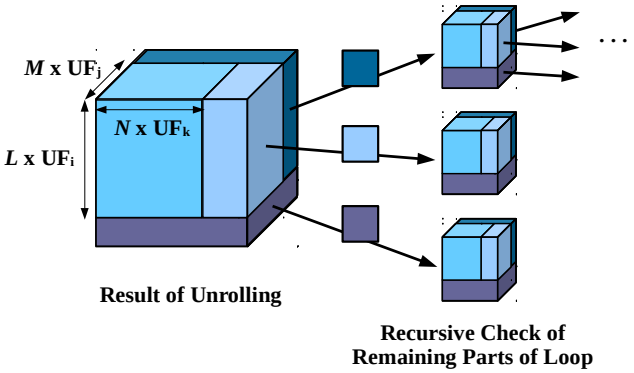


Fig. 5: Recursive exploration of unrolling factors.

Algorithm 1 illustrates the process. The inputs to the algorithm are a loop nest with known iteration bounds and information about CGRA architecture such as the number and capabilities of the PEs, the number of registers, and the latency of each operation. The algorithm first initializes the current and the so-far best unrolling factors and estimates the number of cycles for the original code (lines 8–11). It then invokes the function `unrollLevel` that recursively unrolls each loop level starting with the outermost (level 1) down to the innermost level ($maxLevel$). The loops at each level are unrolled to a empirically determined maximum unrolling factor $maxUF = 8$ (lines 21–24). The expected performance of the unrolled loop nest is estimated when the innermost loop has been reached (lines 25–26). Lines 28–31 find the best unrolling factors for the loops that process the remaining iterations from from $\lfloor iter/UF \rfloor$ to UF as described above. The unrolling factors are stored along with the estimated number of cycles whenever a better combination has been found (lines 33–35).

3) Cycle Estimation: The targeted CGRA supports predicated execution [17] and does not require separate prologue or epilogue code before and after the software-pipeline loop. The performance in cycles of a loop mapped to CGRA as a function of the iterations n is then given by

$$cycles_{cgra}(n) = n * II + (SC - 1) * II \tag{2}$$

where SC, the stage count, is the length of one loop iteration in cycles divided by II. To estimate the cycles of an unrolled loop without computing the modulo schedule, we approximate Equation 2 as follows. First, the MII is used as an approximation of II. The MII can be calculated using the number of PEs, the type of instructions they can execute, and the instructions of the loop body if it were to be unrolled. This gives the cycle estimation formula:

$$est\_cycles_{cgra}(n) = n * MII + (SC - 1) * MII$$

The number of iterations n is known thanks to prior constant propagation, and the stage count SC can be approximated by dividing the longest path in the loop body by MII. In general, the stage count is much lower than the iteration count, and since the absolute accuracy of the model is not a concern as long as the relative performance is estimated correctly, we omit the term with the stage count altogether.

$$est\_cycles_{cgra}(n) = n * MII$$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2018.2857278, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

Algorithm 1 Find Optimal UF

```
1:  Input:
2:      loopNest: loop nest code with iteration bounds
3:      arch: CGRA architecture (#PEs, PE instr., instr. latency)
4:  Return:
5:      UFs: best unrolling factors for each loop nest
6:      cycles: estimated num. cycles for unrolling
7:  function FINDBESTUNROLLING(loopNest, arch)
8:      maxLevel ← depth of loopNest
9:      best.UFs ← [1, 1, 1, ..., 1]  // maxLevel entries
10:     curr.UFs ← [1, 1, 1, ..., 1]  // maxLevel entries
11:     best.cycles ←ESTIMATECYCLE(loopNest, arch, best.UFs)
12:     UNROLLLEVEL(loopNest, arch, 1, maxLevel)
13:     return best.UFs, best.cycles
14:
15:  Input:
16:     loopNest, arch: as before
17:     level: current loop level
18:         maxLevel: innermost loop
19:          1: outermost loop
20:     maxLevel: number of levels
21: function UNROLLLEVEL(loopNest, arch, level, maxLevel)
22:     for uf = 1 to maxUF do
23:         curr.UFs[level] ← uf
24:         unroll loop at level level by uf
25:         if level = innermost then
26:             cycles ←ESTIMATECYCLE(loopNest, arch, curr.UFs)
27:
28:             for each remaining loop do
29:                 UF_rem, cycle_rem ←
                        FINDBESTUNROLLING(loopNest, arch)
30:                 Save UF_rem for curr.UFs
31:                 cycles ← cycles + cycles_rem
32:
33:             if cycles < best.cycles then
34:                 best.UFs[level] ← uf
35:                 best.cycles ← cycles
36:         else
37:             UNROLLLEVEL(loopNest, arch, level + 1, maxLevel)
```

This estimates the cycles of the innermost loop. For a loop nest, the iterations of the outer loops must also be considered. We multiply the estimated cycles of the innermost loop by the number of iterations of all outer loop and get

$$est\_cycles_{total} = MII \times iter_{total}$$

where $iter_{total}$ represents the product of all loop iterations counts of a loop nest.

4) Search Space Pruning: The search space of the exhaustive recursive UF search is very large, as every UF combination of the main loop and any generated remaining loops are considered. To allow the UF search to complete in reasonable time pruning is applied.

With pruning, the search space is reduced by an MII-based threshold ($maxMII$) rather than the UF-based threshold ($maxUF$) in Algorithm 1 line 22. This is because the MII of a loop has a more direct relationship to the execution cycles than the UF itself. When the MII of the unrolled loop exceeds $maxMII$, unrolling at the current level stops continues at the next nesting level. We empirically set maxMII to twice the MII of the innermost loop at an unrolling factor of 1 (no unrolling).

5) Estimation Accuracy: An important question is whether the cycle estimation model, despite its simplifications, is accurate enough to guide the exploration of the unrolling factor space. Figure 6 plots the estimated versus the actual values for the MII (of the innermost loop) and the total execution cycles of a loop nest of the `GEMM` function.

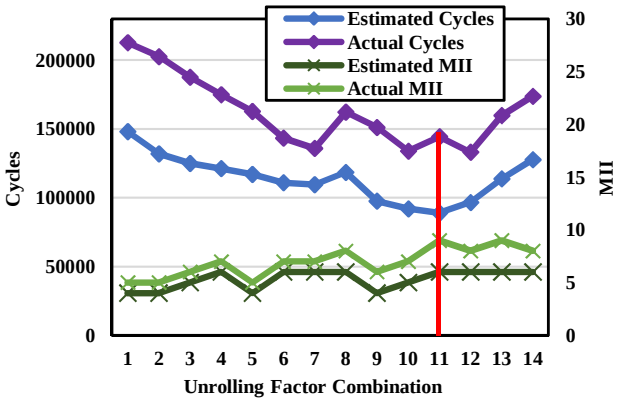


Fig. 6: Estimated vs actual cycles/MII for a loop nest.

The x-axis shows different samples in the UF search space as the search algorithm progresses. In general, the estimated and the actual MII and execution cycles show a good correlation. In the figure, the unrolling factors of combination 11 are chosen because the lowest number of estimated cycles. In this particular case, the model would not pick the configuration with the best performance, however, the actual cycles of the chosen combination are less than 10% higher than the best possible solution.

A similar behavior has been observed for a large number of loop nests; despite its simplifications, the fast cycle-estimation model is a helpful tool to explore the large UF search space.

VI. Experimental Setup

The proposed auto-tuner framework is evaluated using four CNNs of various sizes suitable for embedded systems: LeNet-5 [24], CIFAR10 [25], SqueezeNet [26], and AlexNet [27]. Details of the evaluated networks are given in Table I. The proposed auto-tuning framework has been implemented as a separate optimization phase into the proprietary CGRA modulo-scheduler of Samsung Electronics.

The target CGRA architecture, the Samsung Reconfigurable Processor [16], is a design-time reconfigurable hybrid VLIW/CGRA processor. We evaluate the proposed framework with SRP instances optimized for multimedia workloads. The CGRA, clocked at 500Mhz, comprises between 4 and 16 PEs for the different configurations, 320kB of on-chip SRAM, and a total of 128 32-bit data registers. Figure 7 shows the high-level organization along with the nine different CGRA configurations and the supported operations.

In the 4×4 configuration, for example, there are 8 PEs capable of performing memory operations and 8 PEs that can execute floating point instructions. Figure 7 shows the high-level organization along with the nine architecture variations used to evaluate the CGRA architecture awareness of the approach. PEs capable of executing memory and floating point operations are highlighted as these are the main instruction

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2018.2857278, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

	LeNet	CIFAR	SqueezeNet	AlexNet
Convolutional layers	2	8	26	5
Pooling layers	2	4	4	3
Fully-connected layers	2	0	0	3
Nested Loops	8	20	56	37
Non Nested Loops	6	16	52	37
Unrolled loops	14	36	108	76
Interchanged loops	2	5	23	18
Functions inlined	4	12	30	10
Tuning time (s)	189	65	914	116.6
Rel. code size	2.8	2	2.97	4.19

TABLE I: Properties and optimizations of the CNNs.

types used in DNN processing. CGRAs are mostly unaffected by long-latency operations because of their software-pipelined execution model. Load instructions have a latency of 16 cycles. If larger memories than the 320kB on-chip SRAM are required, 3D-stacked eDRAM can be used in its place. Performance numbers of the SRP were obtained on a cycle-accurate architecture simulator.

The networks are also evaluated on an ARMv8 processor in a Raspberry Pi3 using (1) the same C code that is optimized for the CGRA and (2) the DNN-optimized ARM Compute Library [28] that exploits the SIMD and multi-threading capabilities of the ARMv8 processor.

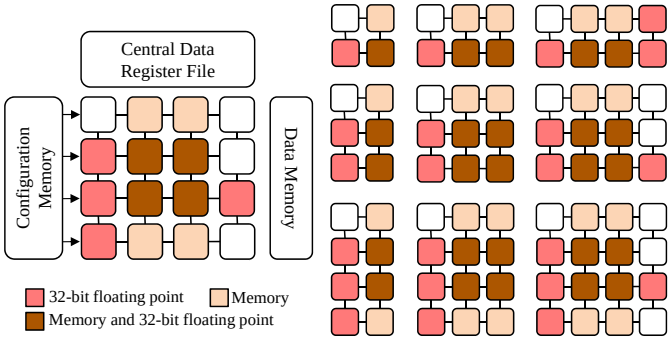


Fig. 7: CGRA overview and configurations.

VII. Performance Evaluation

An overview of the properties of the evaluated networks along with the number of automatically applied optimizations, the tuning time, and the relative code size increase due to unrolling and splitting is given in Table I. For all of the four networks, a large number of optimizations have been applied demonstrating the need for an auto-tuner framework.

A. Performance of Auto-Tuning

To evaluate the effect of the auto-tuner, we compare the baseline to the auto-tuned CGRA code and an optimized execution on the ARMv8 processor. The baseline does not perform any of the optimizations shown in Figure 1 but maps the innermost loop to the CGRA. Figure 8 shows the results. Compared to the baseline, the auto-tuned CGRA code achieves an 11-fold speedup on average with individual performance gains ranging from factor 8.7 to 16.1 for the different networks, demonstrating the efficacy of the presented method. Running the baseline code on the ARMv8 processor, we observe a speedup of over 50 for the CGRA (not

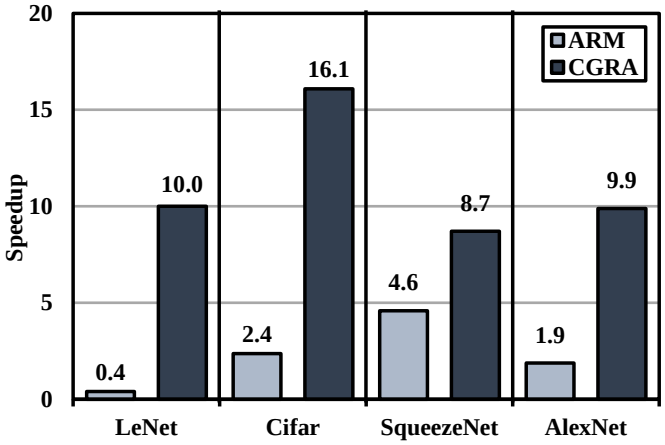


Fig. 8: Speedup Relative to unOpt CGRA

shown in the figure). Exploiting the full capabilities of the ARMv8 processor using SIMD instructions and four threads, the CGRA still outperforms the ARM processor by a factor of 5. The many 1x1 convolutional kernels in SqueezeNet limit the speedup achievable on a CGRA, however, the small register requirements and easy tiling of this CNN are a good match for the ARM processor.

B. Comparison to Other Accelerators

A comparison of the proposed approach using AlexNet and SqueezeNet with different accelerators is shown in Table II. The auto-tuned CGRA execution is compared to a hand-optimized ARMv8 implementation using the ARM Compute Library [28], the Adreno 530 mobile GPU [30], and two dedicated DNN accelerators, the ASIC Eyeriss [6] and Zynq-Net [29], an FPGA implementation.

Comparing computing time, the CGRA with its only 16 PEs achieves comparable performance to other general-purpose accelerators or processors. The Adreno 530 mobile GPU with 256 PEs finishes the task of interference about four times faster (note that, according to [30], inference on the older Adreno 430 takes only 129ms), results compared to ARM are shown in Figure 8. The DNN-specialized FPGA and ASIC accelerators achieve a 6 to 30 shorter computation time. In terms of energy consumption, the CGRA performs particularly well, only being beaten by the heavily optimized Eyeriss. Compared to the ARMv8, the mobile GPU, or the FPGA implementation, the CGRA consumes between 2.5 and 15 times less energy.

Given the general-purpose nature of CGRAs, the results demonstrate that this class of CGRAs can be be a viable alternative for acceleration in existing systems alongside mobile GPUs.

C. CGRA Configuration Adaption

To show that the proposed auto-tuning framework is able to consider and exploit the architectural features of a given CGRA architecture, we compare the baseline with auto-tuned code for 9 different CGRA architectures ranging from 2x2 up to 4x4 PEs. Figure 9 shows that the speedup increases

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2018.2857278, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

	ZynqNet [29]	Adreno 530 [30]	SRP [16]	Cortex-A53 [31]	Eyeriss [6]
Type	FPGA	mobile GPU	CGRA	CPU (ARMv8)	ASIC
SoC/board	Zynq XC-7Z045	Snapdragon 820	-	BCM2837	-
Inference (ms)					
AlexNet	-	-	990	5211	29
SqueezeNet	158	207.1	829	1573	-
Energy/Img (mJ)					
AlexNet	-	-	149	1250	8
SqueezeNet	1880	569	124	380	-
Power (W)	12	2.748	0.15	0.24 [32]	0.278
Tech.	28nm	14nm	32nm	20nm	65nm
PEs	-	256	16	4	168
Clock (MHz)	200	624	500	1200	200
Buffer Size (kB)	256	-	320	-	108
Reg. mem. (kB)	-	-	0.25	-	84
Precision	32b float	32b float	32b float	32b float	16b fixed

TABLE II: Comparison with different accelerators.

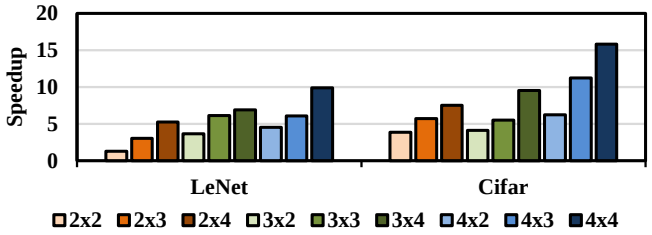


Fig. 9: Cycle ratio of various CGRA architectures.

with the number of PEs. Both evaluated networks show similar speedups. In the 2×2 configuration, automatic optimization achieves a limited speedup because the only 4 PEs do not leave much room for improvement.

D. Effect of Individual Optimizations

The effect of the successive optimizations phases on performance are shown in Figure 10 for the CIFAR network. The speedup, the number of multiply-accumulate (MACC) and memory operations are shown for each optimization phase relative to the unoptimized code using `GEMM` (G) for every kernel. `OptCGRA` (oC) shows the speedup for kernels where `optCGRA` is selected. As designed, the main advantage of `optCGRA` over GEMM is a significant reduction in memory operations.

Figure 11 shows a histogram of the chosen unrolling factors for the different loop nests of SqueezeNet. The framework determined an UF vector of {1,1,1}, i.e., no unrolling, for only four of the well over 100 loops, demonstrating the necessity of an automated framework.

The overhead of the framework is shown in terms of auto-tuning time and the increased code size in the last two rows of Table I. The tuning process takes between 1 and 15 minutes depending on the network structure, or about three orders of magnitude faster than a human. If auto-tuning framework is currently implemented using only one thread; if auto-tuning time becomes an issue, the UF parameter search space could easily be parallelized. Generating separate functions for each

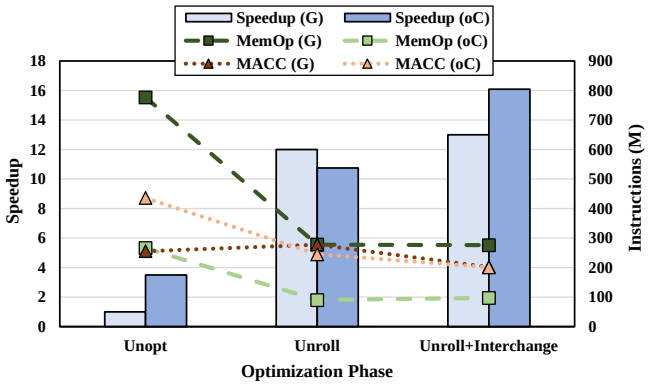


Fig. 10: Optimization Effect

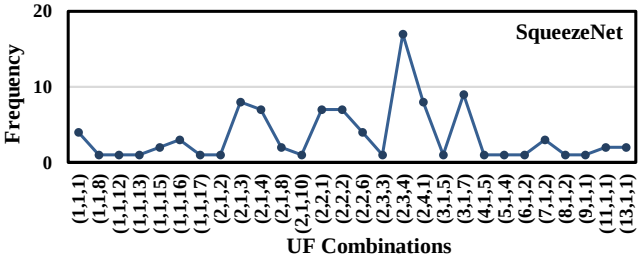


Fig. 11: Distribution and Frequency of UFs

layer incurs a code size increase from 2 to 4 times which is acceptable in comparison to the achieved speedup values.

VIII. Related Work & Discussion

A. Related Work

There are several other techniques for optimizing DNNs on embedded hardware, these have typically taken the form either of FPGA implementations or dedicated ASICs, although a small number of CGRA based approaches also exist. Benchmarks of some of these approaches as well as ours can be seen in Table II. Regarding previous work on CGRA acceleration of CNNs, [33] [33] present a CGRA as CNN accelerator. However here a custom CGRA architecture is designed using SRAM and static scheduling of instructions to PEs, meaning

that this approach is heavily dependent on both the CGRA design and the specific CNN structure. What is proposed in this paper is an automatic framework that can adapt to various CGRA architectures to fully utilize them. This allows the proposed approach to effectively use existing CGRAs as low power CNN accelerators.

B. Discussion

The results of this study show that general-purpose loop accelerators such as CGRAs can be a viable alternative to dedicated DNN processing hardware. In particular, applications requiring general-purpose acceleration for video and audio content can benefit from this approach. This is also particularly useful for games that monopolize the main processor and the mobile GPU. In addition, systems with lower power requirements without graphics acceleration, such as those used for biomedical applications [11] can also profit from using CGRAs for CNN acceleration. For DNNs implemented with low-precision numbers, FPGA implementations can be tailored to arbitrary precitions. ARM and CGRA implementations can equally profit from SIMD support; we thus expect similar results for computations using 16 or 8-bit precision. CGRAs, however, do have some limitations. First, we assumed that all memory can be stored on-chip. The memory size is a design parameter and can be increased; if this is not possible, the proposed algorithms can be tiled and data copied to/from the off-chip memory in smaller chunks. Second, the CGRA used for this study has only 16 PEs which is only a fraction what mobile GPUs or most DNN accelerators have. While it is possible to generate larger CGRAs with, for example, 8x8 PEs, existing modulo schedulers do not scale well to architectures larger than 8x8. One option is to use two CGRAs that run in parallel, however, if that much processing power is needed, a dedicated DNN accelerator is likely the better choice. Nevertheless, the use of CGRAs to accelerate CNNs is an attractive alternative for low-power embedded devices that also need to process different data-intensive tasks such as smartphones, tablets, or cameras.

IX. Conclusion

This paper proposed an automatic auto-tuner framework that optimizes CNN code for execution on a CGRA. The framework analyzes the CNN structure and the CGRA architecture and applies a copy propagation and a number of loop transformation to generate an efficient mapping of the CNN on the CGRA. To explore the large parameter space of unrolling nested loops, a performance estimation model for CGRAs in employed. The auto-tuner framework is able to improve the performance of the CGRA-mapped code by a factor of 11. Compared to other accelerators, the CGRA achieves similar performance and significantly lower energy consumption, demonstrating the potential of existing CGRAs as effective CNN accelerators and reducing the need for dedicated hardware.

Acknowledgments

We thank the anonymous reviewers for their comments and suggestions. This work was supported in part by BK21 Plus for Pioneers in Innovative Computing (Dept. of Computer Science and Engineering, SNU) funded by the National Research Foundation (NRF) of Korea (Grant 21A20151113068), the Basic Science Research Program of NRF funded by the Ministry of Science, ICT & Future Planning (Grant NRF-2015K1A3A1A14021288), the Neural Processing Research Center of the Samsung Advanced Institute of Technology, and by the Promising-Pioneering Researcher Program through Seoul National University in 2015. ICT at Seoul National University provided research facilities for this study.

References

[1] J. M. Kanter and K. Veeramachaneni, "Deep feature synthesis: Towards automating data science endeavors," in *IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, Oct 2015, pp. 1–10.

[2] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[3] Google Ltd., "Tensor processing unit (TPU)," http://cloud.google.com/blog/big-data/2017/05/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu, 2017, (online; accessed February 2018).

[4] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15, 2015, pp. 92–104.

[5] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016, pp. 1–12.

[6] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16, 2016, pp. 367–379.

[7] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16, 2016, pp. 267–278.

[8] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel paterns," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17, 2017, pp. 389–402.

[9] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "ADRES: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix," in *13th International Conference on Field Programmable Logic and Applications (FPL)*, 2003, pp. 61–70.

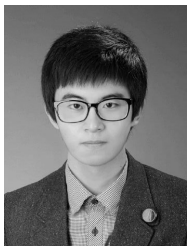
[10] M. Quax, J. Huisken, and J. van Meerbergen, "A scalable implementation of a reconfigurable wcdma rake receiver," in *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 3*, ser. DATE '04, 2004, pp. 230–235.

[11] C. Kim, M. Chung, Y. Cho, M. Konijnenburg, S. Ryu, and J. Kim, "ULP-SRP: Ultra low power samsung reconfigurable processor for biomedical applications," in *International Conference on Field-Programmable Technology (FPT)*, 2012, pp. 329–334.

[12] J. Lee, Y. Shin, W.-J. Lee, S. Ryu, and K. Jeongwook, "Real-time ray tracing on coarse-grained reconfigurable processor," in *International Conference on Field-Programmable Technology (FPT)*, Dec 2013, pp. 192–197.

[13] W.-J. Lee, S.-H. Lee, J.-H. Nah, J.-W. Kim, Y. Shin, J. Lee, and S.-Y. Jung, "SGRT: a scalable mobile gpu architecture based on ray tracing," in *ACM SIGGRAPH 2012 Posters*, 2012, p. 44.

[14] Y. Park, H. Park, and S. Mahlke, "Cgra express: Accelerating execution using dynamic operation fusion," in *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, ser. CASES '09, 2009, pp. 271–280.
[15] Samsung Electronics, "Samsung Exynos 4 Quad (Exynos 4412) User's Manual," Exynos_4_Quad_User_Manaul_Public_REV1.00-0.pdf, 2012. [Online]. Available: http://www.samsung.com/global/business/semiconductor/file/product/
[16] D. Suh, K. Kwon, S. Kim, S. Ryu, and J. Kim, "Design space exploration and implementation of a high performance and low area coarse-grained reconfigurable processor," in *International Conference on Field-Programmable Technology (FPT)*, 2012, pp. 67–70.
[17] S. A. Mahlke, R. Hank, J. McCormick, D. I. August, and W.-M. W. Hwu, "A comparison of full and partial predicated execution support for ilp processors," in *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ser. ISCA '95, 1995, pp. 138–150.
[18] B. Kågström, P. Ling, and C. van Loan, "Gemm-based level 3 blas: High-performance model implementations and performance evaluation benchmark," *ACM Trans. Math. Softw.*, vol. 24, no. 3, pp. 268–302, Sep. 1998. [Online]. Available: http://doi.acm.org/10.1145/292395.292412
[19] D. Coppersmith and S. Winograd, "Matrix multiplication via arithmetic progressions," *Journal of Symbolic Computation*, vol. 9, no. 3, pp. 251 – 280, 1990.
[20] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling," *IEE Proceedings - Computers and Digital Techniques*, vol. 150, no. 5, pp. 255–261, 2003.
[21] T. Oh, B. Egger, H. Park, and S. Mahlke, "Recurrence cycle aware modulo scheduling for coarse-grained reconfigurable architectures," in *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES '09, 2009, pp. 21–30.
[22] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Transactions on Mathematical Software*, vol. 16, no. 1, pp. 1–17, Mar. 1990. [Online]. Available: http://doi.acm.org/10.1145/77626.79170
[23] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, ser. MICRO 27. New York, NY, USA: ACM, 1994, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/192724.192731
[24] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov 1998.
[25] J. Redmon, "Darknet: Open source neural networks in c," http://pjreddie.com/darknet/, 2013–2017.
[26] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and ¡1mb model size," *CoRR*, vol. abs/1602.07360, 2016.
[27] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12. USA: Curran Associates Inc., 2012, pp. 1097–1105. [Online]. Available: http://dl.acm.org/citation.cfm?id=2999134.2999257
[28] ARM Ltd, "Arm compute library v18.05," 2018. [Online]. Available: https://github.com/ARM-software/ComputeLibrary
[29] D. Gschwend, "Zynqnet: An fpga-accelerated embedded convolutional neural network," Master's thesis, Swiss Federal Institute of Technology Zurich (ETH-Zurich), 2016.
[30] M. Motamedi, D. Fong, and S. Ghiasi, "Fast and energy-efficient CNN inference on iot devices," *CoRR*, vol. abs/1611.07151, 2016.
[31] ARM Ltd, "Arm cortex-a53 processor," 2012. [Online]. Available: https://developer.arm.com/products/processors/cortex-a/cortex-a53
[32] A. Frumusanu and R. Smith, "Arm a53/a57/t760 investigated," 2015. [Online]. Available: https://www.anandtech.com/show/8718/the-samsung-galaxy-note-4-exynos-review/4
[33] M. Tanomoto, S. Takamaeda-Yamazaki, J. Yao, and Y. Nakashima, "A cgra-based approach for accelerating convolutional neural networks," in *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, Sept 2015, pp. 73–80.



Inpyo Bae Inpyo Bae received his bachelor's degree in computer science and engineering at Seoul National University, Korea in 2016. He is a master's Student in the Computer Systems and Platforms Laboratory at the same university. He is interested in deep neural network processing and compiler optimization. More information can be found at http://csap.snu.ac.kr/members/inpyo.

Barend Harris Barend Harris is a master's student at Seoul National University in the Computer Systems and Platforms lab. He received his bachelor's degree in computer science at the University of Groningen in the Netherlands. His research interests include deep learning and mobile applications. More information can be found at http://csap.snu.ac.kr/members/barend.

Hyemi Min Hyemi Min is a master's student at Seoul National University in the Computer Systems and Platforms lab. She received her bachelor's degree in management information systems at Chungbuk National University. She is interested in deep neural network processing. More information can be found at https://csap.snu.ac.kr/members/hyemi.

Bernhard Egger Bernhard Egger received the diploma in computer science from the Swiss Federal Institute fo Technology (ETH) in Zurich, Switzerland and the PhD degree in computer science and engineering from Seoul National University, Korea. He then spent three years at Samsung Electronic's research institute before re-joining Seoul National University in 2011 where is currently is an associate professor at the School of Computer Science and Engineering. His research interests include programming language design, compilers, operating systems, and architectures for mobile and heterogeneous manycore systems. He is a member of the IEEE and ACM. More information can be found at http://csap.snu.ac.kr/bernhard.