2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO) | 978-1-6654-6272-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/MICRO56248.2022.00044

GCD^2: A Globally Optimizing Compiler for Mapping DNNs to Mobile DSPs

Wei Niu
William & Mary, USA
wniu@wm.edu

Jiexiong Guan
William & Mary, USA
jguan@wm.edu

Xipeng Shen
North Carolina State University, USA
xshen5@ncsu.edu

Yanzhi Wang
Northeastern University, USA
yanz.wang@northeastern.edu

Gagan Agrawal
Augusta University, USA
gagrawal@augusta.edu

Bin Ren
William & Mary, USA
bren@wm.edu

***Abstract*—More specialized chips are exploiting available high transistor density to expose parallelism at a large scale with more intricate instruction sets. This paper reports on a compilation system *GCD*^{2}, developed to support complex Deep Neural Network (DNN) workloads on mobile DSP chips. We observe several challenges in fully exploiting this architecture, related to SIMD width, more complex SIMD/vector instructions, and VLIW pipeline with the notion of *soft dependencies*. *GCD*^{2} comprises the following contributions: 1) development of matrix layout formats that support the use of different novel SIMD instructions, 2) formulation and solution of a global optimization problem related to choosing the best instruction (and associated layout) for implementation of each operator in a complete DNN, and 3) SDA, an algorithm for packing instructions with consideration for *soft dependencies*. These solutions are incorporated in a complete compilation system that is extensively evaluated against other systems using 10 large DNN models. Evaluation results show that *GCD*^{2} outperforms two product-level state-of-the-art end-to-end DNN execution frameworks (TFLite and Qualcomm SNPE) that support mobile DSPs by up to 6.0× speedup, and outperforms three established compilers (Halide, TVM, and RAKE) by up to 4.5 ×, 3.4 ×, and 4.0 × speedup, respectively. *GCD*^{2} is also unique in supporting real-time execution of certain DNNs, while its implementation enables two major DNNs to execute on a mobile DSP for the first time.**

***Keywords*-VLIW instruction packing; compiler optimization; deep neural network; mobile devices;**

I. Introduction

Despite the upcoming end of Moore's law, the last several years have seen a quick increase in transistors density. For example, in going from 22 nm technology to 10 nm, Intel chips saw a nearly 7× increase in transistor density, and the most chip manufacturers are building chips with more than 100 million transistor per square millimeter at the time of writing this paper[1]. All processors, but more particularly the specialized ones, have exploited this density by supporting an increasing amount of parallelism, often combined with intricate ways in which this parallelism can be exploited. Even in mainstream processors, the SIMD width has increased and the flexibility of programming API has improved with AVX-512 instruction set that has features like *scatter*, *gather*, and *masks*.

An example of a class of specialized chips that offer a programming interface suited for general purpose processing is the Digital Signal Processing (DSP) chips. Particularly, smartphones have invested in sophisticated DSP chips that are also capable of accelerating other highly parallel workloads. To date, however, there is only a limited exploration on the use of DSP chips for other workloads [1], [2], [3], [4].

In recent years, machine learning (ML) or deep learning (DL) workloads, particularly the Deep Neural Networks (DNNs), have emerged as important workloads that have been targeted on a range of hardware – from mainstream processors and accelerators [5], [6], [7], [8], [9], [10], [11], [12] to mobile devices [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23] (including mobile DSP [24]) to chips specifically designed for them [25], [26], [27], [28]. A particular requirement, and the driver of our work, is performing inference using complex Deep Neural Network (DNN) models on mobile phones in a time, memory, and power-efficient manner. We observe that DSP chips are a natural candidate for accelerating DNN inference in a mobile setting, not only because mobile phone already have a DSP chip, but also because these chips are optimized for matrix and vector computations on fixed-point values.

This paper reports a compilation system that optimizes Deep Neural Networks (DNNs) for execution on a mobile DSP chip. As a quick motivation for this effort, results from Table I show that with an existing framework, TFLite [14], execution on a DSP chip outperforms both mobile CPU and GPU in terms of execution time and power. Conceptually, however, it also turns out that compiling for the DSP chip involves dealing with many advanced features, especially with respect to low-level parallelism exposed through its instruction set, requiring techniques well beyond the ones implemented in current systems or otherwise developed. More specifically, modern (mobile) DSP chips have much more complex SIMD instruction sets with both a larger

[1]https://www.techcenturion.com/7nm-10nm-14nm-fabrication

978-1-6654-6272-3/22/$31.00 ©2022 IEEE
DOI 10.1109/MICRO56248.2022.00044

Authorized licensed use limited to: William & Mary. Downloaded on June 01,2023 at 02:21:04 UTC from IEEE Xplore. Restrictions apply.

Table I: **Latency and Power Comparisons among Mobile CPU, GPU, and DSP.** Experiments are conducted on a Samsung Galaxy S20 with TFLite [14]. CPU, GPU, and DSP uses int8, float16, and int8, respectively. Power is collected by the Android system interface. Results are for each inference.

Model	#MACS†	Latency (ms)			Power⋆		
		CPU	GPU	DSP	CPU	GPU	DSP
EfficientNet-b0 [33]	0.4G	53	11.3	9.1	10.7	1.6	1.0
ResNet [34]	4.1G	62	34.4	13.9	6.2	2.3	1.0
PixOr [35]	8.8G	280	64.6	43	6.7	1.8	1.0
CycleGAN [36]	186G	4320	477	450	5.5	1.2	1.0

† #MACS denotes the number of multiply-accumulate operations.
⋆ Normalized by DSP's power for readability.

width and a greater variety of instructions as compared to the mainstream processors, and thus require techniques beyond those explored in current literature [29], [30], [31]. Besides 1024-bit width, there are instructions combining vector operations and reductions in different ways, going even beyond Intel's additions under VNNIW and FMAPS extentions [32]. In addition, VLIW instructions exist that can combine multiple SIMD instructions for simultaneous execution, and there are other performance characteristics that require new methods for effective mapping of the workload.

This paper develops techniques for exploiting these architectural features. Our contributions include:

- **Methods for Exploiting Disparate SIMD Instructions.** We develop data layouts and execution schemes that use different new instructions for key Deep Learning (DL) kernels. We also investigate the trade-offs between different approaches depending upon the size of the operands.
- **Formulating and Solving a Global Optimization Problem.** We show how the choice of instruction (and their corresponding data layouts) for one operator impacts the choice for their successor and formulate a global optimization problem. We show an optimal linear-time solution for this problem when the operators form a linear chain, and develop useful heuristics for the general case of a computational graph.
- **VLIW Packing (i.e., Scheduling) Problem.** Considering many unique aspects of our target architecture (including the notion of *soft* dependencies, and latency sensitivity), we present a novel Soft Dependencies Aware (SDA) algorithm for instructions packing.
- **Design of an End-to-End Compilation System.** We engineer a system that includes a nuanced code generation design and several additional optimizations.

GCD^2 is extensively evaluated on 10 real-world large DNNs, with a range of model sizes and operator counts and designed for various ML tasks, targeting popular mobile DSPs. Compared with two state-of-the-art DNN frameworks (TFLite [14] and Qualcomm SNPE [37]) that support end-to-end mobile DSPs execution, *GCD*^{2} achieves 2.8× and 2.1× speedup (in geometric mean), respectively, reaching *real-time execution* for some of them for the first time. In fact, for two of the models, *GCD*^{2} implementation supports mobile DSP execution for the first time. While comparing with three established compilers (Halide [38], TVM [5], and RAKE [4]) that support efficient kernels execution on mobile DSPs, *GCD*^{2} achieves 4.5×, 3.4×, and 4.0× speedup, respectively. *GCD*^{2} outperforms others primarily because of improved SIMD execution and optimized VLIW instruction scheduling and the evaluation justifies the choices made in *GCD*^{2}'s algorithms for these optimizations.

II. Executing DNNs on Mobile DSPs

Modern mobile DSPs have become increasingly powerful with key features as follows: 1) larger SIMD widths, 2) richer vector instructions with growing computation capabilities, and 3) more flexible instruction pipelines that can tolerate certain data dependencies. Take Qualcomm Hexagon 698 DSP [39][2] as an example. Its SIMD width is 1024-bit, twice that of Hexagon 680 [40] and its instruction set includes multiple SIMD/vector instructions (e.g., `vmpy`, `vmpa`, and `vrmpy` elaborated in Section III), and can support complicated MAC (multiply–accumulate) operations. Multiple vector (and scalar) instructions can be packed into a VLIW pipeline, further improving the computational throughput. Finally, the pipeline offers hardware mechanisms to guarantee execution correctness even in the presence of certain dependencies, thus offering more flexibility.

Mobile DSPs support fix-point operations (8/16/32-bit) with extremely high performance (e.g., the theoretical peak for Hexagon 698 DSP is 15 TOPS [41]). While considering DSP chips for DNN execution, the important context here is that *Quantization*, a well-known technique to convert floating-point values to integer ones, has been very effective in accelerating DNN executions, particularly on resource-constraint devices [42], [43]. The cutting-edge DNN acceleration frameworks, (e.g., TFLite [14] and SNPE [37], and Qualcomm's built-in library Hexagon NN [44]) aim to combine the benefits of both quantization and mobile DSPs to accelerate DNN execution, achieving both (near) state-of-the-art model accuracy and lower latency as compared to the other parts of the mobile SoC (i.e., CPUs and GPUs). Similarly, MobiSR schedules the super-resolution model over Heterogeneous Mobile Processors (including CPU, GPU, and DSP) [45].

Despite these rapid developments, compilers and libraries built for DSP chips cannot fully exploit the device's computation power – this applies to, but is not limited to, the compilers and libraries for DNN execution listed above. Specifically, the performance of the mobile DSP is sensitive to 1) the input/output data layout, and 2) the VLIW instruction packing

[2]Qualcomm Snapdragon is one of the most popular SoC and many generations of Snapdragon are equipped with Hexagon DSPs. Although our presentation and evaluation is on Hexagon DSP, the work is generally applicable to other mobile DSPs as well, e.g., Cadence, which is the other major player in the mobile DSP market.

Authorized licensed use limited to: William & Mary. Downloaded on June 01,2023 at 02:21:04 UTC from IEEE Xplore. Restrictions apply.

Table II: **Execution Latency w/ Different SIMD Instructions (and Layouts) for Matrix Multiplication $C = A \times B$.** M, K, and N denote the dimension size of Matrix A ($M \times K$), B ($K \times N$), and C ($M \times N$), respectively. Execution latency and total data size with padding are normalized by `vmpy` for readability. Smaller numbers mean better latency or less padding. Bold ones denote the best case.

M	K	N	Execution Latency			Total Data Size w/ Pad		
			`vmpy`	`vmpa`	`vrmpy`	`vmpy`	`vmpa`	`vrmpy`
32	32	32	1.00	0.79	**0.63**	1.00	0.56	**0.33**
64	64	64	1.00	**0.69**	0.76	1.00	**0.60**	**0.60**
96	96	96	1.00	1.06	**0.89**	1.00	1.00	**0.82**
128	128	128	**1.00**	1.10	1.23	**1.00**	**1.00**	**1.00**

(or scheduling) in view of all hardware resource constraints. This is because first, various SIMD/vector instructions are designed to perform MAC operations in different ways and they are friendly to different input/output data sizes and data layouts. Second, the VLIW pipeline imposes many constraints on the instructions that can be packed together.

In the context of DNN acceleration, complex DNN designs challenge the DSP-oriented implementations in multiple ways. First, modern DNNs usually consist of many operators (e.g., the latest BERT consists of over 1000 operators [46]), and even with the same operator, operands can be of different shapes and sizes. Mapping growing SIMD/vector width and instruction set (variety) to these operators and operands is challenging. Second, as discussed above, the complex opportunities and constraints in VLIW packing need to be considered for implementations of specific operators.

III. Instructions and Layouts

Our target instruction set includes novel and complex SIMD instructions capable of optimizing computations found in ML (and scientific) workloads. We show three representative instructions in Figure 1. While these instructions are used for multiple operators in a DNN (e.g., the convolutions), our presentation here uses matrix multiplication for illustration. Similarly, other instructions like `vtmpy` and `vmpye` can also be used to implement these operators. Our discussion here considers only three instructions. However, as a motivation, we first show the trade-offs between their use.

Table II shows how the cost of matrix multiplication varies with the three choices when input tensors have different shapes. We can see that the instruction `vmpy` (and the corresponding `1-column` layout, both are elaborated later) provides better execution efficiency if the operands have a certain length. However, for other cases, this instruction causes padding overheads, thus making the other instructions more time- and space -efficient.

As additional background, many recent works show that the floating-point representations (and operations) for weights and activations are not necessary to achieve good accuracy for DNNs, but instead fixed point (8-bit or even less) suffices [42], [47], [48], [49], [50], [51]. However, one caution is that the product between two 8-bit values should be stored in 16-bits to avoid data overflow, and similarly, accumulating several such products requires 32-bits. In either case, a requantization phase is required to generate the 8-bit final output.

With this motivation and background, we explain the existing instructions and associated data layouts we have developed. In Figure 1 (a), we show the instruction `vmpy`, whose inputs are a vector with 128 8-bit values and four scalar values. In `vmpy`, four consecutive values in the vector are multiplied by four distinct scalars, with the output being two vectors with 64 16-bit values, each storing alternate results of multiplications.

In Figure 1 (b), the input for the instruction `vmpa` are two vectors with 128 8-bit values each. A pair of corresponding values from the two vectors are multiplied by two scalar values and then added together. Specifically, alternate pairs are multiplied with the first two and the last two scalars, respectively, and accumulated to two different output vectors.

Finally, in Figure 1 (c), the instruction `vrmpy` is illustrated – here, four consecutive values from the vector are successively multiplied by four distinct scalar values, and accumulated together. The result is a vector with 32 32-bit values.

In this work, we have developed novel dense matrix data layouts that optimize the use of these instructions for multiple key operators in DNN computations (e.g., `MatMul`, `CONV`, `Depthwise CONV`, etc.), and this part takes matrix multiplication (`MatMul`), a critical kernel for our target workload as an example. Developing layouts for implementing arbitrary loop nests using these or similar instructions is an open problem beyond the scope of this paper.

In Figure 2 (a), we show the layout that enables the use of `vmpy` instruction shown earlier in Figure 1 (a). For efficiency, it is very important that the set of values that are to be loaded to or stored from a vector register are stored in a contiguous fashion. The layout we use is referred to as the `1-column` layout. The numbers shown in the boxes represent the offset of the location of that element. In `1-column` layout, a set of 128 rows is stored in a column-major way, and this pattern is repeated for the next set of rows. In carrying out the matrix multiplication, the first column is loaded to a vector and all values are multiplied with the first weight (0) stored in the scalar register. The outputs are two vectors storing 64 16-bit elements each, which will eventually be shuffled to obtain an output layout matching the input layout. The process continues by loading the next 128 elements physically stored in our layout, multiplying them with the second weight (1), and reducing the output to the same two vectors.

In Figure 2 (b), we show the layout and the key steps of matrix multiplication with the instruction `vmpa`, which was shown earlier in Figure 1 (b). The layout we have designed is referred to as `2-column` layout – within the

Authorized licensed use limited to: William & Mary. Downloaded on June 01,2023 at 02:21:04 UTC from IEEE Xplore. Restrictions apply.

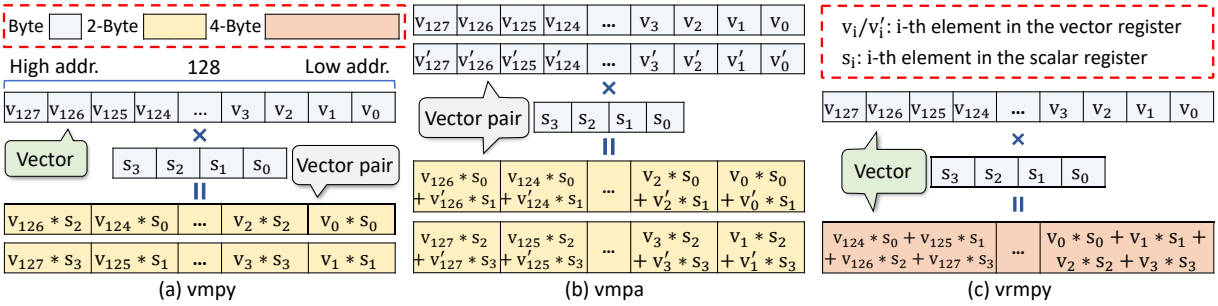
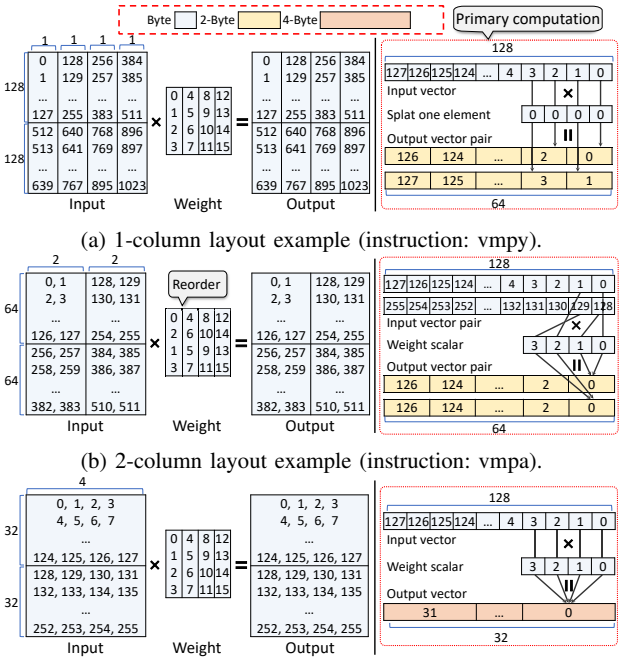


Figure 1: **SIMD/Vector Multiply Instruction Examples in Mobile DSP Chip**



(a) 1-column layout example (instruction: vmpy).

(b) 2-column layout example (instruction: vmpa).

(c) 4-column layout example (instruction: vrmpy).

Figure 2: **Data Layouts to Support Usage of Varied SIMD Instructions for Matrix Multiplication.** Each number denotes the linear storage offset of an element. A blue, yellow, and orange cell takes 1, 2, and 4 bytes, respectively. Left shows data storage, and right shows computation.

64-row panels, values for 2 columns are stored adjacent to each other, before following the column-major storage. In applying matrix multiplication, elements 0, 1, 128, and 129, which are from the same rows of the matrix, are multiplied by the output weights 0, 1, 2, and 3, respectively, stored in scalar registers. Note that the two corresponding output elements in the output vectors need to be further added to obtain the results.

In Figure 2 (c), we show the matrix multiplication operations and layouts with the use of the instruction `vrmpy` shown in Figure 1 (c). The input and weight matrix are of different shapes as compared to the previous examples, in order to illustrate the layout and the computation. Here,

Table III: **SIMD Instructions Selected and Performance by RAKE [4] and *GCD*^{2}.** Representative `Conv2d` kernels (w/ varied shapes, 7×7, 1×1, and 3×3) are from ResNet-50.

Conv2d properties			Instruction		Speedup
Input shape	Weight shape	Output shape	RAKE	Ours	Ours/RAKE
1x3x224x224	64x3x7x7	1x64x112x112	vrmpy	vmpy	1.63x
1x64x56x56	64x64x1x1	1x64x56x56	vmpy	vmpa	1.98x
1x128x28x28	128x128x3x3	1x128x28x28	vrmpy	vmpy	2.06x

panels of 32 rows are used and four elements from each row are stored together. Four elements in a row are multiplied with four weights stored in scalar registers. We also note that while there is an instruction somewhat similar to `vrmpy` in Intel instruction set (`vpdpbusd`), there are no counterparts to `vmpy` or `vmpa` at the current time.

Overall, our work considers a relatively small number of candidate instructions for implementing a single operation, using a "pre-designed" approach for each pair of operator and instruction. Efforts do exist on trying to automate the selection of instruction and code generating using the instruction [3], [4], [52]. We have conducted a brief comparison of our approach against the code generated by the most recent of these efforts (which also targets the same instruction set), i.e. RAKE [4]. As shown in Table III, our approach is able to deliver significantly higher performance. Thus, while automation of instruction selection and code generation is valuable, current approaches are not matching the "pre-designed" approach we are taking.

IV. System Design of *GCD*^{2}

This section highlights the major optimizations developed in *GCD*^{2}, followed by a brief summary of implementations.

A. SIMD Global Opt. Problem Formulation

From the discussion earlier in Section III, the important takeaway is that different instructions can be used for the same operation, but with different requirements on input formats, resulting in different output formats, and with different trade-offs (which were summarized earlier in Table II).

With a relatively small number of instructions available to implement a single operation, the instruction and the layout selection can be performed (in isolation) by explicitly

Authorized licensed use limited to: William & Mary. Downloaded on June 01,2023 at 02:21:04 UTC from IEEE Xplore. Restrictions apply.

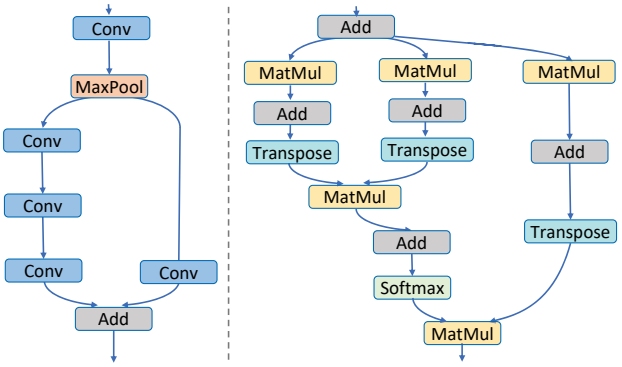


Figure 3: **Examples of Computational Graphs.** Left and right show partial CGs in ResNet [34] and TinyBERT [53].

considering all choices and choosing the one that requires the fewest cycles for execution. However, it turns out that with distinct input and output layouts for different instructions, choices for each operation cannot be made in isolation. Suppose an operator A can be implemented in the most efficient fashion using the instruction `vmpa`. Let the output of the operation A be the input to the operation B. Without considering the need for the formatting of input tensors, let the most efficient implementation of B be using the instruction `vrmpy`. However, the output tensor from the operation A will be in the two-column format (Figure 2 (b)), whereas if B is implemented using `vrmpy`, it is expected that the input tensors are in the four-column format (Figure 2 (c)). Converting the layout of a tensor itself is a time-consuming step. Thus, if the sequence of two operators A and B are considered, it is possible that the most efficient implementation involves using the same instructor (and thus layouts) for the two operators. In practice, DNN models use many operators (e.g., the model EfficientDet-D0 used in our evaluation has 822 operators), and thus, we have a complex optimization problem.

To formulate this global optimization problem, we use an existing intermediate representation called the Computational Graph (CG) [5], which captures the data-flow and basic operator information like the operator type and parameters. Figure 3 shows examples of such graphs. Let V be the set of vertices in a CG and let E be the set of edges. Each vertex is an operation that produces exactly one output tensor. A directed edge (v_i, v_j) denotes that the output of the vertex (operation) v_i is (one of) input(s) to the operation v_j. The source of the edge e is also denoted as $vin(e)$ and similarly, the destination of e is denoted as $vout(e)$.

Now, given an operator (vertex) $\mathscr{O}$ in the CG, let it have a set of immediate predecessors we denote as $Pre(\mathscr{O})$. By each predecessor, we denote interchangeably both the operators and their output. After performing the local analysis of possible implementations and associated layouts for the operator $\mathscr{O}$ we obtain a set of possible execution plans $EP(\mathscr{O})$, comprising execution plans $ep_1(\mathscr{O})$, $ep_2(\mathscr{O})$, and so on. Associated with every execution plan, there is a cost of execution, denoted by $Cost(ep_i(\mathscr{O}))$, which is based on the number of instructions (cycles) required. This cost calculation assumes that all input tensors are already stored in the required layout for the SIMD instruction used.

We consider an execution plan $ep_i(\mathscr{O})$ and a predecessor tensor of $\mathscr{O}$, which we denote as $\mathscr{I}$ ($\mathscr{I} \in Pre(\mathscr{O})$). If the operator $\mathscr{I}$ is executed with the plan $ep_j(\mathscr{I})$, then there could be a data transformation with the associated cost $TC(ep_j(\mathscr{I}), ep_i(\mathscr{O}))$ (this cost will be 0 when data transformation is not required).

Given this background, the global optimization problem is as follows. For each operator (vertex) v in the CG, we want to select an execution plan ep_v, such that the total cost of execution for the graph G, which is denoted as

$$Agg\_Cost(G) = \sum_{v \in V} Cost(ep_v(v)) + \sum_{e \in E} TC(ep_{vin(e)}(vin(e)), ep_{vout(e)}(vout(e))) \quad (1)$$

is minimized. In the expression above for $Agg\_Cost(G)$, the first term is the cost of execution associated with each operation under the choice of plan made, whereas the second term is the cost of data transformation between the layouts for the source and the sink of the edge, under the choices of implementation plans chosen for the source and sink operators.

B. Layout & Instruction Select Solution

It is easy to see that a trivial approach for solving this problem will involve comparing $k^{|V|}$ options, where $|V|$ is the number of vertices in the graph and k is number of (assumed fixed for all operators) options available for each operator. Even when k is 2 or 3, this cost can be easily prohibitive for realistic DNN models. Furthermore, the above problem is really a Partitioned Boolean Quadratic Programming (PBQP) problem, which is known to be NP-hard [54].

If we simply have a linear chain of operations $O_1, O_2, \ldots, O_n$, then the following approach can be used to solve the problem. Let $Sol(i, j)$ denote the lowest possible cost of execution operations $O_1, O_2, \ldots, O_i$ such that the output from the operator O_i is the j^{th} available choice ($j \leq k$, where k is the number of choices available for each operator). Then, we have

$$Sol(i, j) = min_{l=1,\ldots,k}(Sol(i-1, l) + TC(ep_l(O_{i-1}), ep_j(O_i)) \quad (2)$$

Here, $Sol(i, j)$ is computed by comparing k choices, which are the lowest cost ways of reaching each of the k different output formats for the previous operation in the chain. It is easy to see that this recurrence can be solved in $O(|V| \times k^2)$ time. Moreover, this solution can be easily extended to the cases when either every path from a "source" vertex of a DAG to a given vertex is of the same length, or when every vertex has at most one output. However, this approach does not work for an arbitrary DAG and we focus on an effective

Authorized licensed use limited to: William & Mary. Downloaded on June 01,2023 at 02:21:04 UTC from IEEE Xplore. Restrictions apply.

heuristic solution. While considering a PBQP solver [54], [55], which is not guaranteed to provide an optimal solution but is in practice close, is an option, we instead focus on exploiting the properties of our target domain. For this, we consider the following definition:

Definition IV.1 (Cost Optimal Partitioning)**.** Given a computational graph G, a cost optimal partitioning of G is a disjoint graph partitioning $P = \{G_1, G_2, \ldots, G_n\}$, such that for $Agg\_Cost(G)$ (as defined in Equation 1), we have

$$Agg\_Cost(G) = Agg\_Cost(G_1) + Agg\_Cost(G_2) + \ldots + Agg\_Cost(G_n) \quad (3)$$

Note that we use the popular definition of graph partitioning, where the edges between vertices that are in different partitions are not considered part of either partition. If such partitioning can be found, the optimal plans for all operators within each partition can be determined in isolation, translating to a significant reduction in the complexity of search.

In practice, What we can hope to achieve is to find a set of partitions that can be optimized independently, i.e. where the lowest cost for the entire graph is achieved by choosing plans within each partition independently. To achieve this, we note that an edge $e = (v_i, v_j)$ is a desirable partitioning edge if 1) the node v_j has only one predecessor (v_i), and 2) The operator v_j is a *layout transformation operator* or the transformation along the edge e is a *profitable* transformation. Typical examples of layout transformation operators include `Reshape` and `Transpose` – they do not perform any computations but change the shape of the operand. A transformation along an edge is considered *profitable* if the reduction in execution time of the successor operator with the transformed layout is higher than the cost of the data transformation itself. The intuition for this definition of desirable partitioning edge is that decisions on nodes leading up to this edge and vertices following this edge can be made in isolation.

However, as next challenge for us, partitioning a graph typically involves many *cut edges*. Now, if we can find a cut edge that is *dominant*, i.e, if every path from the (assumed to be unique) source vertex in the DAG to the (again, assumed unique) sink vertex passes through this edge, then the problem is simplified. When this is not feasible, we can add *complementary edges* to the identified cut edges to create complete partitions.

C. VLIW Optimization

Instruction packing or scheduling is a long-standing issue in VLIW research that has been proved to be NP-hard [56], [57]. Because of the specific opportunities as well as challenges associated with our target architecture, a new algorithm is developed in this work.

Optimization Foundation: Hard/Soft Dependencies. For our target architecture, dependencies between two instructions

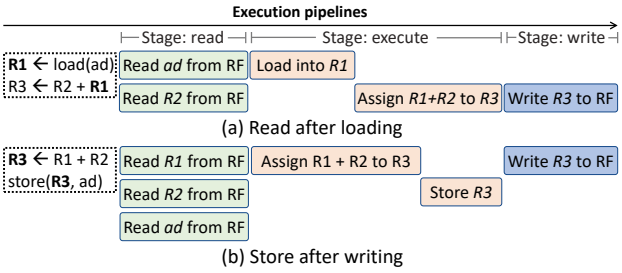


Figure 4: **Two Examples of Packing Instructions with Soft Dependencies.** Different colors show different VLIW execution pipeline stages (`read` in green, `execute` in orange, and `write` in blue). In (a), the second stage (`Assign R1+R2 to R3`) of the second instruction requires to wait for the completion of the first instruction, incurring packing penalty. A similar situation happens to (b) between `Assign` and `Store`.

can be characterized into two types with respect to their implication on placing them in the same VLIW packet[3]

- *Hard dependency* denotes a strict dependence relationship where placing such instructions in the same packet likely produces incorrect results.
- *Soft dependency* denotes a relaxed dependence relationship, and placing instructions together produces correct results; however, the resulted execution performance is likely degraded to a certain degree. An example of the soft dependency in our target architecture is the one between a scalar addition operation and a consumer of the result of such an addition.

To further illustrate the nuances associated with soft dependencies, we show two examples of packing instructions with soft dependencies in Figure 4. Figure 4 (a) shows a dependency between a load operation and an arithmetic operation that consumes the loaded value. Each of these two instructions takes 3 clock cycles[4] individually, however, if they are packed in the same packet, they can execute correctly by taking 4 cycles in total[5]. This example shows that packing instructions with soft dependencies together takes less clock cycles than not packing them together at all (i.e., treating the soft dependency as a hard dependency). However, if sufficient number of instructions are available without any dependencies between them, we will prefer to not pack instructions with soft dependencies together. Figure 4 (b) shows a similar example with a soft dependency between an arithmetic operator and a store operation. Which

[3]This classification is independent of the traditional classification of dependencies into flow/RAW, output/WAW, and anti/WAR, though soft dependencies can only be RAW or WAR.

[4]According to the target microarchitecture design, each VLIW pipeline execution comprises three stages, `read` from Register File (RF), `execute`, and `write` to RF, though some of these stage can be be empty. Our explanations will assume that each stage is 1 clock cycle [58].

[5]Mobile DSP processors (e.g., Hexagon DSPs) execute instructions within each VLIW packet in parallel, but without overlap between packets.

Authorized licensed use limited to: William & Mary. Downloaded on June 01,2023 at 02:21:04 UTC from IEEE Xplore. Restrictions apply.

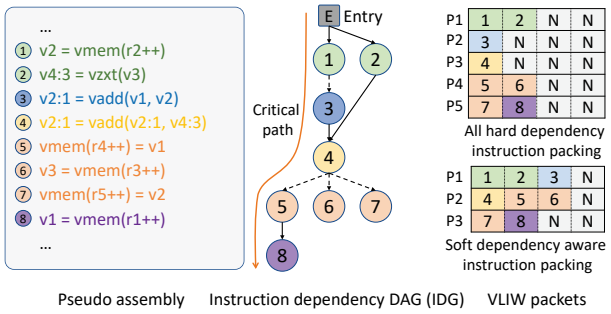


Figure 5: **An Instruction Packing Example.** The left part shows part of the pseudo assembly code for the innermost nested loop performing 2D Element-wise Addition: `R = A + B + C`, where `A`, `B`, and `C` are two-dimensional uint8 arrays and `R` is a two-dimensional int16 array. v2:1 denotes a 16-bit register combining 2 8-bit registers v2 and v1. The middle part shows an IDG, in which, solid edges denote hard dependency, dot edges denote soft, and critical path is colored in red. Right shows the packing results from our solution and an sub-optimal solution that treats all soft dependencies as hard (*soft to hard*). N denotes an empty instruction slot.

dependencies are soft and which ones are hard depends upon the microarchitecture. This information needs to be obtained from the details of processor implementation (e.g., [58]) and made available to the instruction packing algorithm.

Soft Dependencies Aware (SDA) VLIW Packing Algorithm. Because of the notion of soft dependencies, we have developed a new VLIW instruction packing algorithm. Besides handling the distinction between soft/hard dependencies, the algorithm is cognizant of other constraints. While a packet can have up to 4 instructions, there can be a limited number of slots for each type of instruction. As an example, packing two shift operations together is not allowed. This instruction packing is implemented as an additional optimization step of LLVM's assembly code generation.

Like much of the previous work, the packing algorithm uses the notion of a *critical path* [59]. and its overall goal of minimizing execution time as two sub-goals: 1) reducing the total number of instruction packets, and 2) packing instructions with identical or similar latency together to minimize VLIW pipeline stalls. The work also has many similarities with algorithms for code generation targeting superscalars, in the sense the goal is to minimize intra-bundle RAW stalls [60].

Our presentation of the algorithm (as shown in Algorithm 1) is supported by the running example from Figure 5. Its left hand side shows the pseudo assembly code for a part of an innermost nested loop of a frequently occurring `Add` operator in deep neural networks (`R = A + B + C`), where `A`, `B`, and `C` are two-dimensional uint8 arrays and `R` is a two-dimensional int16 array. Take the instruction of $v2:1 = vadd(v1, v2)$ in this pseudo assembly code as an example. v1 and v2 are two 8-bit registers. v2:1 denotes a

Algorithm 1: Soft-dependency-aware VLIW Packing

```
   Func packing: instructions ← [Packet]
1  cfg ← build_cfg(instructions)
2  all_packets ← Stack()
3  foreach block in cfg.block do
4      idg ← build_IDG(block)
5      free_insts ← Set()
6      find_free_instruction(idg, free_insts)
7      while free_insts is not empty do
           /* Build critical path from IDG */
8          critical_path ← get_critical_path(idg)
9          cur_packet ← critical_path[-1]
           /* Iterate all the free instruction */
10         while len(cur_packet) ¡ 4 do
               /* Select the most profitable
                  instruction */
11             inst ← select_instruction(free_insts, cur_packet)
12             find_free_instruction(idg, free_insts)
13             if inst is None then
14                 break
15             else
16                 cur_packet.add(inst)
17                 idg.remove(inst)
18         all_packet.add(cur_packet)
19 return all_packets

   Func select_instruction: free_insts, packet ← Instruction
20 all_insts ← resource_constraint(free_insts, packet)
21 if all_insts is empty then
22     return NULL
23 hi_lat ← highest_latency(packet)
24 best ← NULL
25 foreach i in all_insts do
       /* The criteria of profitability */
26     i.score ← (i.order + i.pred) ×w - abs(hi_lat - i.lat) ×(1 − w)
27     if soft_dependency(i, packet) then
28         i.score ← i.score - p(i, packet)
29     if best is NULL or best.score ¡ i.score then
30         best ← i
31 return best
```

16-bit register combining 2 8-bit registers (v2 and v1) to store the addition result.

Returning to our algorithm, it first builds a Control-Flow Graph (CFG) on assembly for each operator, and finds the basic block corresponding to the computation kernel of each operator (usually the largest basic block). Next, it builds an instruction dependency DAG (called IDG) based on the hard/soft dependency information, and finds the critical path with the longest execution latency. The middle part of Figure 5 shows the IDG – here, a vertex represents an instruction, and an edge represents the dependence between two instructions. A solid edge represents a hard dependency and a dotted edge represents a soft dependency. Take the instructions (or vertices) 4, 5, 6, and 7 in this figure as an example. The dependencies between the instructions 4 and 5, 4 and 6, and 4 and 7 are all soft dependencies. IDG also contains an artificial entry vertex. The number shown with the vertex corresponds to the assembly instruction in the left. The critical path is colored in red. The vertices with identical

Authorized licensed use limited to: William & Mary. Downloaded on June 01,2023 at 02:21:04 UTC from IEEE Xplore. Restrictions apply.

colors have the same rank (distance to the entry).

Based on the IDG and the critical path, the algorithm now packs instructions. When creating a new packet, the algorithm always uses the last (unpacked) instruction in the critical path as a seed (line 9). Next, such an instruction is packed with other instructions that either do not have any outgoing edges or have only *soft-dependence edges* to an instruction to be packed (all of these instructions are called *free* instructions). This step consists of three major sub-steps: i) iterating through all free instructions (line 7), 2) finding a candidate instruction from the set of free instructions (line 11), and 3) grouping the candidate instruction into the current packet. Particularly, the key second sub-step (i.e., finding a candidate instruction) comprises of two steps: first, for the current working packet, the algorithm finds all instructions that can be packed while meeting the hardware constraints (line 20), and also determines the highest latency (*hi_lat*) among the instructions that are already in the current packet; second, it iterates these available instructions to pick up the *best* instruction and returns it (lines 25 to 30). Note that, the *best* instruction selection is based on this instruction's *score* (*i.score*) that is calculated as follows:

$$i.score = (i.order + i.pred) \times \mathrm{w} - abs(hi\_lat - i.lat) \times (1 - \mathrm{w}) \quad (4)$$

According to Equation 4, the score of an instruction is decided by its three attributes, its distance from the entry node (*i.order*), its predecessor instruction count (*i.pred*), and its latency (*i.lat*). The first two have positive impacts on the score, while the absolute difference between this instruction's latency (*i.lat*) and the latency of the longest instruction already in the current packet (*hi_lat*) has a negative impact on the score. The former is because it is desirable to include instructions that have a longer chain of dependencies and/or a total large number of instructions that it is dependent on. The latter, on the other hand, wants to create more efficiency by packing instructions of the same (or very similar) latency values together. This algorithm introduces two new parameters (w is short of weight, and p is short for penalty) that are empirically decided. w aims to control the weight of the three factors' impact (line 26), while p aims to control the impact of *soft dependency* on this packing (line 28). Specifically, the value p depends both on the instruction i under consideration and the instructions already placed in the *packet*, and captures the stall that the soft dependence will cause. For comparison in our experiments, we also create a version of our algorithm that reduces all soft dependencies to 'none' or no dependence – this version of the algorithm will ignore the calculation of this penalty. Next, to complete the description of our algorithm, after one packet is created, the algorithm repeats by finding the critical path of the remaining sub-graph.

Returning to Figure 5, the right part shows the packets after scheduling (N denotes an empty slot). This example compares our Soft Dependencies Aware (SDA) packing algorithm (bottom) with a sub-optimal algorithm (called *soft to hard*) that treats all soft dependencies as hard ones (top). Taking the first seed (vertex 8, i.e., the last instruction in the critical path) as an example. 8 and 6 cannot be packed together (because of hardware constraints) and 8 can be packed with 7. Our packing algorithm can continue to explore the packing opportunity between 4 and 5/6 because 5 and 6 only have soft dependencies to 4, and the soft dependencies allow the packing for 1, 2, and 3; however, these opportunities do not exist in *soft to hard* version of the algorithm. In summary, our algorithm delivers a schedule with only three packets, while the sub-optimal *soft to hard* version generates a schedule with two additional packets. Evaluation results in Section V further validate our algorithm's efficacy.

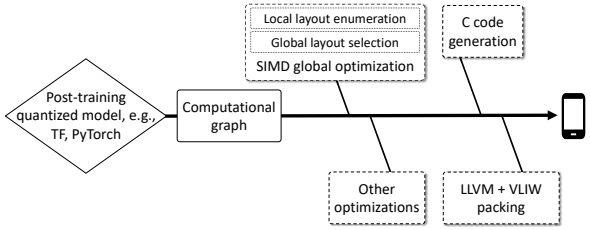


Figure 6: **System Workflow of *GCD*^{2}.**

Impact of Unrolling. Loop unrolling plays an important role in the schedule quality by affecting the scheduling scope and the register pressure. Different from previous work like [61], *GCD*^{2} employs a low-cost heuristic solution specifically designed for DNN operators. The basic idea is to perform a fast *adaptive* unrolling setting selection according to the shape of output tensors, for example, for GEMM, different unrolling settings are designed for varied output shapes (skinny, near-square, and fat). Our empirical study in Section V proves that this approach outperforms some simple selections while also yielding comparable performance gains to a much more expensive exhaustive search.

D. Putting Everything Together

GCD^2 is implemented on top of an existing end-to-end DNN execution framework, PatDNN [62], [63], [64] to support efficient mobile DSP execution. Figure 6 shows the system workflow of *GCD*^{2}. First, it converts the post-training quantized model to a computational graph (and optimizes it with various techniques, e.g., constant folding) by leveraging the existing framework. Second, it feeds the (optimized) computational graph to the SIMD global optimization module to conduct the local layout (instruction) enumeration and the global layout (instruction) selection. The result here is an optimized SIMD code generation plan including the data layout for each operator and corresponding SIMD instructions to use. This is followed by a pass where other optimizations are applied, e.g., replacing an expensive division operation with a database lookup operation. As the next step, the existing framework and the optimized SIMD code generation plan lead to a "low-level" C code with input/output tensor

Authorized licensed use limited to: William & Mary. Downloaded on June 01,2023 at 02:21:04 UTC from IEEE Xplore. Restrictions apply.

storage details and optimized SIMD intrinsics. Finally, it employs LLVM [65] with our VLIW packing optimization to generate the optimized executable code on the mobile DSP.

V. Evaluation

This section evaluates the performance of GCD^2 by comparing it with five state-of-the-art frameworks, TFLite [14] (V2.6.0), SNPE [37] (V1.55), Halide [38] (V12.0.1), TVM [5] (V0.8.0), and RAKE [4] (V1f99df1). More specifically, TFLite, SNPE, and TVM are the state-of-the-art production-level DNN execution frameworks that can support (or partially support) our target mobile DSP. Both TFLite and SNPE call Hexagon NN, an expert-written hand-tuned library designed by Qualcomm. However, as end-to-end DNN execution frameworks, their computational graph optimizations (graph rewriting, operator fusion, etc.) are different, thus resulting in very different execution performance (as shown in Table IV). Halide, TVM, and RAKE use LLVM as their back-end to generate DSP instructions. They perform packet generation without distinguishing between soft and hard dependencies (i.e., they treat each soft dependency as a hard dependency). It should be noted that Halide, TVM, and RAKE are tensor compilers, while GCD^2 comprises both tensor compiler optimizations (e.g., global data layout optimization) and language compiler optimization (instruction packing). We introduce a version of GCD^2 to facilitate a comparison of tensor compiler aspect of our work with these systems, as we will describe later. Our evaluation has four main objectives: 1) to demonstrate that GCD^2 outperforms all of these state-of-the-art frameworks on mobile DSP (Section V-B); 2) to identify the benefits of specific optimizations and the choices made in our algorithms (Section V-C); 3) to study the power consumption and energy efficiency of GCD^2 against alternative implementations on the same chip (Section V-D); 4) to compare the inference speed and energy efficiency of our mobile DSP-based solution with other embedded DNN accelerators (Section V-E).

A. Evaluation Setup

Models and Datasets. GCD^2 is evaluated on 10 state-of-the-art neural networks (see Table IV) that are categorized into seven groups according to the tasks they perform. Particularly, they include 1) three image classification two-dimensional CNNs (MobileNet-V3 [66], EfficientNet-b0 [33], and ResNet-50 [34]); 2) one image style transfer two-dimensional CNN (FST [67]); 3) one image-to-image translation GAN (CycleGAN [36]); 4) one super resolution two-dimensional CNN (WDSR-b [68]); 5) two object detection two-dimensional CNNs (EfficientDet-d0 [69], and PixOr [35]); 6) one trans-former-based NLP model (TinyBERT [53]); and finally, 7) one transformer-based speech recognition model (Conformer [70]). All the evaluated models in this section are quantized by a standard approach used by well-known TFlite [71](with identical post-training quantization across all frameworks) with 8-bit integers being used for weights and feature maps (activations).

It should be noted that the choice of datasets has a negligible impact on the final inference latency or relative execution speeds, which are the primary metrics in our evaluation. Therefore, and also because of space limitations, we report results from one dataset for each model. MobileNet-V3, EfficientNet-B0, ResNet-50, and CycleGAN are trained on the ImageNet dataset [72], WDSR-b is trained on DIV2K [73], EfficientDet-d0 and FST are trained on COCO [74], PixOr is trained on KITTI [75], TinyBERT is trained on BooksCorpus [46] and English Wikipedia [46], and Conformer is trained on [76]. Because all frameworks employ the identical model quantization approach, they achieve the same accuracy on all models and datasets, and thus accuracy is not reported.

Test Bed. Most of the experiments described in this section are conducted on a Samsung Galaxy S20 (with Snapdragon 865 SoC [39]) that consists of an octa-core Kryo 585 CPU, Adreno 650 GPU, and Hexagon 698 DSP (with Vector eXtensions support). We also tested our framework on older series Snapdragon platforms, which show the similar performance gains against other baseline frameworks. We omit the results due to the space constraints. We note that our optimization designs are general, potentially applicable to other mobile DSP architectures (e.g., Cadence DSPs with increasingly complex SIMD and VLIW supports). All models are executed with their best configurations while the same parameters are used for all execution platforms. Each data involves inferences on 50 different inputs. After excluding the highest/lowest time, an average is taken and reported. As the variation is negligible, ranges are not reported.

B. Comparison with Other Frameworks

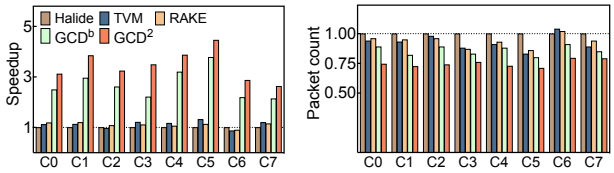
This part evaluates the overall performance of GCD^2 by comparing it against five state-of-the-art frameworks, TFLite, SNPE, Halide, TVM, and RAKE. We compare the performance of GCD^2 with TFLite and SNPE over 10 models. While Halide, TVM, and RAKE have the capability to generate code for the DSP chip, they currently cannot execute full DNN models on this platform. Thus, a `Conv2d` kernel is used for comparison against Halide, TVM, and RAKE.

Execution Latency. Table IV shows the overall performance comparison for all 10 models. TFLite and SNPE do not support Transformer-based models. For the other 8 models, GCD^2 achieves 1.5× to 6.0×, and 1.5× to 4.1× speedup over TFLite and SNPE, respectively. Table IV shows that GCD^2 outperforms TFLite and SNPE mainly because of 1) optimized SIMD instruction selection and layout transformation, and 2) optimized SDA VLIW packing by taking soft dependencies into account. TFLite and SNPE employ a uniform SIMD implementation for each operator type to support mobile DSP execution, and their VLIW

Authorized licensed use limited to: William & Mary. Downloaded on June 01,2023 at 02:21:04 UTC from IEEE Xplore. Restrictions apply.

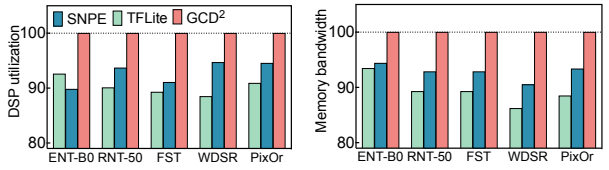
Table IV: **Overall Performance Comparison among TFLite, SNPE, and GCD^2 on Mobile DSP.** "-" means this model is not supported by the framework yet. OverT and OverS are the speedup of GCD^2 over TFLite, and SNPE, respectively. GCD^2's overall ***compilation time*** for these models ranges from 5 minutes (WDSR-b) to 25 minutes (EfficientDet-d0).

Model	Type	Task	#MACS	#Params	#Operators	TFLite (ms)	SNPE (ms)	GCD^2 (ms)	OverT	OverS
MobileNet-V3	2D CNN	Classification	0.22G	5.5M	193	7.5	6.2	**4.0**	**1.9**	**1.6**
EfficientNet-b0	2D CNN	Classification	0.40G	4M	254	9.1	9.2	**6.0**	**1.5**	**1.5**
ResNet-50	2D CNN	Classification	4.1G	25.5M	140	13.9	11.6	**7.1**	**2.0**	**1.6**
FST	2D CNN	Style transfer	161G	1.7M	64	935	870	**211**	**4.4**	**4.1**
CycleGAN	GAN	Image translation	186G	11M	84	450	366	**181**	**2.5**	**2.0**
WDSR-b	2D CNN	Super resolution	11.5G	22.2K	32	400	137	**66.7**	**6.0**	**2.1**
EfficientDet-d0	2D CNN	2D object detection	2.6G	4.3M	822	62.8	-	**26**	**2.4**	**-**
PixOr	2D CNN	3D object detection	8.8G	2.1M	150	43	26.4	**11.7**	**3.7**	**2.3**
TinyBERT	Transformer	NLP	1.4G	4.7M	211	-	-	**12.2**	**-**	**-**
Conformer	Transformer	Speech recognition	5.6G	1.2M	675	-	-	**65**	**-**	**-**
Speedup (geometric mean)									**2.8**	**2.1**



(a) Speedup. (b) Packet count.

Figure 7: **Performance Comparison of GCD^2, Halide, TVM, and RAKE with Individual Kernels.** Left shows the speedup and right shows the packet counts, both normalizing Halide as 1. `Conv2D` operators (from ResNet-50) are used. GCD^b is a sub-optimal version of GCD^2 that contains tensor optimizations only without VLIW packing.



(a) DSP utilization. (b) Memory bandwidth.

Figure 8: **DSP Utilization and Memory Bandwidth Comparison.** These results are as reported by Snapdragon Profiler [77], and normalized with GCD^2.

packing does not consider soft dependencies as GCD^2. It turns out that GCD^2 achieves the most speedup (6.0× over TFLite) on WDSR-b. The reason is that feature map shapes in WDSR vary significantly among different operators, and our instruction selection and layout transformation optimizations deliver much better performance over others.

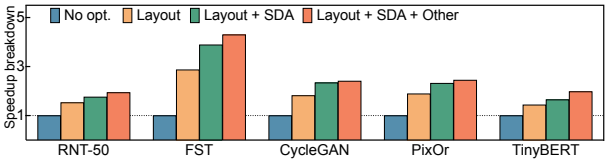
We also note that GCD^2*for the first time* enables mobile DSP execution of two DNNs (TinyBERT and Conformer) because it supports more operators than TFLite and SNPE, e.g., more variants of `MatMul`, and `Pow`. It also *the first time* supports real-time mobile DSP execution of another (EfficientDet-d0).

Next, we compare several individual convolutional computation kernels with Halide, TVM, and RAKE. Because our native compiler optimizations (SDA VLIW instruction packing) built on LLVM can be applied to all other frameworks as well to further improve their performance, we separate tensor compiler optimizations (e.g., our data layout and instruction selection) and native/language compiler optimizations (e.g., SDA VLIW instruction packing) in this comparison by introducing a new version of GCD^2 called GCD^b. GCD^b only contains tensor compiler optimizations, and can be viewed as a more fair comparison against these three tensor compilers. In this comparison, the first 8 unique `Conv2D` operators in ResNet-50 are used. Figure 7 (a) and (b) show the speedup and the packet count for these 8 `Conv2D` kernels, respectively, and all results are normalized by Halide. It turns out that GCD^2 outperforms Halide, TVM, and RAKE with significant speedups due to both its layout optimizations and VLIW instruction packing. In comparing GCD^b with other tensor compilers, GCD^b achieves up to 3.8×, 2.7×, and 3.3× over Halide, TVM, and RAKE due to tensor compiler optimizations like layout and instruction selection. In addition, our instruction packing algorithm results in fewer numbers of packets (25% < Halide, 19% < TVM, and 21% < RAKE on average, respectively). Please also refer to Section V-C for a more detailed performance breakdown study.

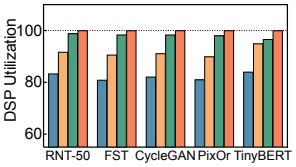
Overall Performance Analysis. To further understand the performance difference among above frameworks, Figure 8 compares DSP utilization and memory bandwidth. This experiment uses 5 representative models out of 8 supported by both TFLite and SNPE, including EfficientNet-B0 (ENT-B0), ResNet-50 (RNT-50), FST, WDSR, and PixOr. Experiments on other models show similar trends and are excluded because of space limits. The data is collected from Snapdragon Profiler [77]. For DSP utilization, TFLite and SNPE can only achieve 88% to 93%, and 89% to 95% of GCD^2's utilization, respectively. For memory bandwidth, TFLite and SNPE can only utilize 86% to 93% and 90% to 94% of GCD^2's, respectively. These results show GCD^2 better utilizes mobile DSP's computing and memory resources with better VLIW instruction pipeline execution and higher SIMD parallelism.

It should be noted that the theoretical peak performance of Hexagon 698 reported by Qualcomm is 15 TOPS [41].

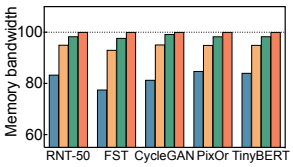
Authorized licensed use limited to: William & Mary. Downloaded on June 01,2023 at 02:21:04 UTC from IEEE Xplore. Restrictions apply.



(a) Speedup over the baseline (No opt.)



(b) DSP utilization.



(c) Memory bandwidth.

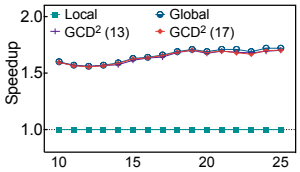
Figure 9: **Performance Breakdown Analysis.** Speedup over the baseline (normalized with the no-opt version). DSP utilization and memory bandwidth analysis (both normalized with the GCD^2 optimal version as 100%). The results are collected from Snapdragon Profiler [77].

However, this number includes its Neural Processing Unit that is not publicly programmable yet. To get the peak performance of the publicly available vector processing unit (HVX), we test the highly optimized matrix multiplication kernel in the Qualcomm Hexagon SDK with small inputs that can fit into the L-1 cache, and achieve the performance of 3.7 TOPS. Our evaluation shows GCD^2 achieves up to 1.51 TOPS for an individual layer in DNN inference. Considering the necessary data loading and memory latency costs involved, this value shows effective practical use of the hardware.

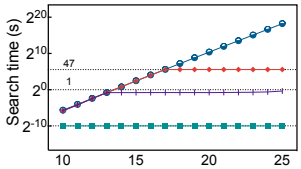
C. Impact of Opt. and Algorithmic Features

Impact of Different Optimizations. To understand how different optimizations (instruction and layout selection, VLIW packing, and other optimizations) contribute towards performance speedups, Figure 9 (a) studies the impact of these optimizations with 5 representative models that cover 2D CNN, GAN, and Transformer (EfficientNet-B0 (ENT-B0), ResNet-50 (RNT-50), FST, WDSR, and PixOr). We evaluate each compiler-based optimization speedup incrementally over our baseline (w/o proposed optimizations). Compared with `No opt`, instruction and layout selection brings 1.4× to 2.9× gains, VLIW scheduling achieves additional 1.2× to 2.0× speedup, and finally, other optimizations (e.g., replacing an expensive division operation with a database lookup) add 1.1× to 1.4× speedup. Figure 9 (b) and Figure 9 (c) further reveal that instruction and layout selection also has the largest impact on DSP utilization and memory bandwidth.

Instruction (and Layout) Selection Analysis. This section justifies the choice we have made in performing global layout selection. Specifically, we compare the algorithm used in GCD^2 with two baselines - `local optimal` and exhaustive search based `global optimal` solutions. The local optimal solution selects the layout with the best performance independently for each operator, whereas the



(a) Speedup.



(b) Search time.

Figure 10: **Layout Optimization Analysis.** X-axis denotes the number of operators in the computational graph. The left figure shows the speedup over local optimal with different numbers of operators. The right figure shows the search time, and its y-axis is logarithmically scaled.

global optimal always conducts an (expensive) exhaustive search on the entire computational graph to find the optimal solution.

For the purpose of these experiments, partial computational graphs are extracted from ResNet-50 using contiguous operators. Figure 10 (a) compares the model execution performance among local optimal, global optimal, and our two versions – GCD^2 `(13)` and GCD^2 `(17)` mean the maximum number of operators within each sub-graph is 13, and 17, respectively. Compared with local optimal, GCD^2 brings 1.55× to 1.7× speedup, while global optimal brings 1.56× to 1.72× speedup. This validates the design choice we have made – specifically, the performance of GCD^2 `(13)` is almost identical to global optimal. At the same time, it is clear that local-only decisions impose large data transformation overheads and do not achieve good performance.

Figure 10 (b) compares the search time for the four solutions. Obviously, the search time in global optimal solution increases exponentially, making it impracticable even when there are 25 operators (complete models have more operators, see Table IV). The search time is over 80 hours with only 25 operators in the graph, while GCD^2 `(13)` and GCD^2 `(17)` need less than 2 seconds and 1 minute, respectively.

VLIW Packing Analysis. One of the unique aspects of our SDA VLIW instruction packing is the treatment of *soft* dependencies. We evaluate this by comparing our method against two versions: 1) all soft dependencies are treated as hard dependencies, i.e., separating all instructions with soft dependencies into different packets (`soft to hard`; 2) all soft dependencies are treated as no dependencies `soft to none` (i.e., removing lines 27, 28 in Algorithm 1 and thus not associating with penalty with packing an instruction with a soft dependency). Figure 11 reports the effectiveness of our optimization using 5 models and establishes our current algorithm does better than either of these choices. GCD^2 achieves up to 2.1×, and 1.4× speedup compared with `soft to hard` and `soft to none`, respectively because of better packing efficiency as compared to `soft to hard` and fewer runtime stalls as compared to `soft to none`.

Unrolling Analysis. Figure 12 (a) shows the performance

Authorized licensed use limited to: William & Mary. Downloaded on June 01,2023 at 02:21:04 UTC from IEEE Xplore. Restrictions apply.

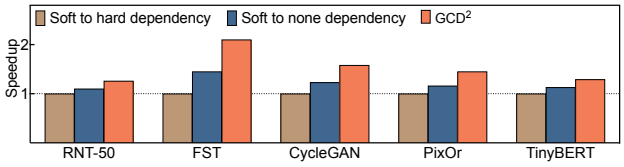
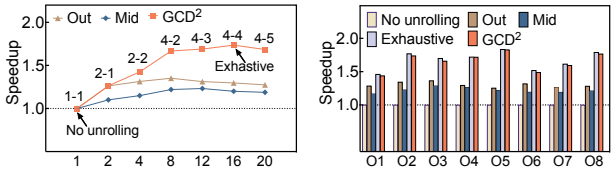


Figure 11: **VLIW Scheduling Analysis.** The version treating all soft dependencies as hard ones is used as the baseline.



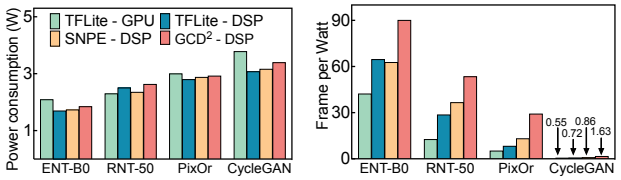
(a) On a single kernel. (b) On multiple kernels.

Figure 12: **Unrolling Factor Analysis on a Single MatMul Kernel and on Multiple MatMul Kernels.** The x-axis in the left figure denotes the unrolling factors. The right figure shows the performance comparison among the best settings of three unrolling strategies (Out, Mid, and GCD^2) on 8 operators (from O1 to O8). For comparison, it also shows versions w/o unrolling and w/ exhaustive search.

comparison of different unrolling strategies for a matrix multiplication kernel (three loop-levels): `Out` (only unroll the outer-most-level loop), `Mid` (only unroll the mid-level loop), and `Exhaustive` (unroll the loops by an exhaustive search). We omit the inner-most-level loop as a possibility as vectorization is performed at that level. The x-axis denotes the unrolling factor, while the speedup is normalized by no unrolling, i.e., when the unrolling factor is 1. The unrolling settings of GCD^2 for both loop levels are also labeled in this figure. The best configuration by exhaustive search is $4-4$. GCD^2 achieves higher performance compared with the other two options. For all options, we see the expected result that the performance drops if unrolling factor is too large due to increasing register spilling. Figure 12 (b) compares the performance of `Out`, `Mid`, `Exhaustive` search, and GCD^2 under different matrix multiplication kernels – here again the y-axis is normalized by `No unrolling` in each kernel. Unrolling factor in `No unrolling` is 1, while `Out` and `Mid` both use the best unrolling factor obtained from Figure 12 (a). Compared with exhaustive search (`Exhaustive` that searches the best unroll plan for a loop structure in some common unrolling configurations), GCD^2 achieves very comparable performance while saving significant search time (exhaustive search generally takes over 3 minutes for each kernel). GCD^2 unrolling achieves much higher performance compared with the other two strategies across all kernels.

D. Power Consumption and Energy Efficiency

Figure 13 compares the total power consumption and energy efficiency of GCD^2 against TFLite and SNPE also



(a) Power consumption. (b) Energy efficiency: infer. frames/W.

Figure 13: **Comparison of Total Power Consumption (left) and Energy Efficiency in Inference Frames/Watt (right).** Three DSP frameworks and TFLite with GPU back-end on 4 representative DNNs.

Table V: **Inference Speed and Energy Efficiency Comparison with ResNet-50 on EdgeTPU [78] and NVIDIA Jetson Xavier [79].** FPS is short for frames per second, and FPW represents for inference frames per Watt.

Platform	Device	FPS	Power	FPW
EdgeTPU [78]	Edge TPU (int8)	17.8	2 W	8.9
Jetson Xavier [79]	GPU + DLA (fp16)	291	≈30 W	9.7
Jetson Xavier [79]	GPU + DLA (int8)	1100	≈30 W	36.7
GCD^2	DSP (int8)	**141**	**2.6 W**	**54.2**

executed on DSP (`*-DSP`) on four representative DNN models (EfficientNet-b0, ResNet-50, PixOr, and CycleGAN). As additional baseline, TFLite on a mobile GPU, Qualcomm Adreno 650 GPU on the same Snapdragon 865 SoC (`TFlite-GPU`) is also included. Figure 13 (a) shows the total power consumption of each solution, where we see that `TFLite-GPU` consumes the most power (ranging from 2.1 Watt to 3.8 Watt), and three DSP-based solutions consume less power. GCD^2-DSP consumes less power than `TFLite-GPU` (by around 3.6% on average) while consuming slightly higher power than `TFLite-DSP` and `SNPE-DSP` (7.2% and 6.7% on average, respectively). GCD^2-DSP consumes more power than other DSP solutions mainly because of its better DSP and memory utilization. As this results in reduced execution times, GCD^2-DSP achieves much better energy efficiency as measured in inference frames per Watt – specifically improving on `TFLite-DSP` and `SNPE-DSP` by around 1.7× and 1.5× on average, respectively (Figure 13). Figure 13 also shows that all mobile DSP-based solutions result in better energy efficiency than the state-of-the-art mobile GPU-based solution, `TFLite-GPU`. Specifically, GCD^2 outperforms it by 2.9× in energy efficiency.

E. Comparison with Other DNN Accelerators

To better understand the inference speed and energy efficiency of mobile DSP, we also compare GCD^2 with two popular embedded DNN accelerator-based solutions, EdgeTPU [78] and Jetson Xavier [79] using a representative DNN (ResNet-50). EdgeTPU is a low-power embedded platform with an edge TPU aiming to accelerate integer

Authorized licensed use limited to: William & Mary. Downloaded on June 01,2023 at 02:21:04 UTC from IEEE Xplore. Restrictions apply.

computations. Jetson Xavier utilizes both a GPU and DLA (deep learning accelerator), with operators not supported by DLA executed by the GPU. In this evaluation, EdgeTPU and Jetson Xavier use TFLite, and TensorRT, respectively, as their inference engine. The evaluation results are presented in Table V. Jetson Xavier with int8 results in the highest FPS (frames per second) though with more power consumption. Our mobile DSP solution, GCD^2 achieves 6.1× and 1.48× better energy efficiency (FPW) with the same data type (int8) over EdgeTPU and Jetson Xavier, respectively.

VI. Related Work

This section discusses efforts related to DNN acceleration and compilation, SIMD optimizations, VLIW instruction packing, and other compilation work targeting DSP chips.

DNN Acceleration and ML/DL Compilers. There are many recent efforts on accelerating DNN inference on edge and mobile devices including DeepX [13], TFLite [14], TVM [5], MNN [15], DeepCache [16], DeepMon [17], DeepSense [18], MCDNN [19], and MobiSR [45]. Some of them (e.g., TVM, and TFLite) rely extensively on compiler techniques, and hence are called ML or DL compilers. Most of these efforts do not target DSP, except TVM, TFLite, and MobiSR that offer options to call certain versions of Hexagon NN [44]. They do not focus on SIMD/VLIW optimizations as GCD^2. TASO [9] and AccPar [80] are two recent DNN acceleration efforts with some similarities to GCD^2. TASO's computation-graph-level optimization is restricted to a sub-graph with a limited number of operators, aiming to assist in their proposed effective operator substitution; while GCD^2 focuses on a global optimization aiming to find a data layout solution that can result in the optimized execution of the entire DNN. The partitioning problems considered by AccPar have similarities with the data layout (and instruction) selection problem GCD^2 considered. However, AccPar's formulation is different and can always be solved by dynamic programming, while GCD^2's problem maps to an NP-complete problem, PBQP [54], and thus requires a different solution.

Compiling for DSP Chips. Digital Signal Processing chips have been around for several decades and there have been multiple systems developed for compiling for them [81], [82], [83], [84], including considering SIMD features [85] and exploring VLIW instructions [86], [87]. However, the DSP chip instructions set targeted in this earlier work do not have much correspondence to a modern mobile DSP chip like the one considered in this work. The techniques presented in this work are all related to advances in SIMD instruction sets and properties of VLIW instruction execution. Recently, Ahmad *et al.* [4] have reported a system that does instruction selection and code generation for the same instruction set as the one we have targeted. Their work is more general in considering arbitrary loop nests but does not address the global optimization problem. Moreover, their approach has a high compilation cost, and they report results on small kernels only – our experimental comparison shows better results for our system even on individual operators. The work from Vanhattum *et al.* [3], [52] also has similar focus (and limitations) but their target backend is different, making a direct experimental comparison infeasible. Next, Yang *et al.* have mapped a vision-related DNN to a chip that comprises several DSP processors, performing effective mapping to their vector instruction [2]. However, their work has been applied to a single model and does not include a general compiler-based optimization framework. Prior to that, another system (based on Halide system) was extended to support DSP chips [1], but this work did not emphasize data layout issues.

SIMD Optimizations. Compiler-driven code optimization and generation for SIMD [88], [89], [90] goes back several decades. Earlier work was heavily driven by the fact that Intel SIMD extensions required operands of vector instructions to be contiguous [88], [91], [92]. More advanced techniques in this area used polyhedral models to map arbitrary loop nests for SIMD execution [29], [30] or even consider irregular applications [93]. Because of our target workloads, where there are relatively fewer options for the computations within one operator, but there can be a very long chain of operators, the challenges we address are related to global optimization, and not dealing with arbitrary loop nests. Previous work on global optimizations for SIMD [94], [95] did not consider a comparable instruction set as ours, and therefore, SIMD instruction selection and associated data layout optimizations were not their focus. Recently, Chen *et al.* have developed VeGen [31] that targets the growing diversity in available SIMD and vector instructions. The VeGen compiler extracts what they term as *lane-level parallelism* by finding the instruction most suitable for a loop (nest). This work, however, does not consider the possibility (and costs) of data transformation to use specific instructions, does not target instructions as complicated as the one we have handled, and there are no global optimizations in their work. In another recent work, a JIT compilation system was presented to use Intel SIMD advances for convolution operations [32] – this work, however, does not consider any layout or global optimizations.

VLIW Instruction Packing. VLIW instruction scheduling with timing and resource constraints is a long-standing issue, and many solutions have been proposed for various DSP architectures (that are different from modern mobile DSPs), including advanced software pipelining [96], [97], [98], [99]. Closely related to this work, Six *et al.* [59] discussed a critical path based approach based on a variant of Coffman-Graham list scheduling [100]. This approach is top-down by leveraging the heuristic that instructions with the longest latency path to the exit have priority. However, our scheduling is bottom-up by considering a heuristic of assigning higher priority to instructions that are on a critical path and can enable more instructions packing if they are packed early.

Authorized licensed use limited to: William & Mary. Downloaded on June 01,2023 at 02:21:04 UTC from IEEE Xplore. Restrictions apply.

More importantly, compared with all existing efforts, *GCD*^{2} categorizes data dependencies and tolerates *soft* dependencies with advanced hardware support, and focuses on a more domain-specific design for DNN accelerations on mobile DSP.

VII. Conclusion and Future Work

This paper has presented a compilation system, *GCD*^{2}, for efficiently mapping real-world complex DNN workloads on modern mobile DSP architectures. *GCD*^{2} consists of three major optimizations including the development of matrix layout formats to support novel advanced SIMD instructions in the mobile DSP, a global SIMD optimization procedure that selects optimal SIMD instructions and associated layouts, and an SDA VLIW instruction packing that considers the effect of soft dependencies. *GCD*^{2} is extensively evaluated with ten real-world complex DNNs on popular mobile DSPs. The results show that *GCD*^{2} outperforms two cutting-edge end-to-end DNN execution frameworks supporting mobile DSPs by up to 6.0× and outperforms three established compilers that support efficient computation kernels execution on mobile DSPs by up to 4.5× because of the improved SIMD execution and optimized VLIW instruction scheduling. For certain DNNs, *GCD*^{2} is unique in supporting the real-time execution of the model. For two of these ten models, *GCD*^{2} implementation has, for the first, enabled execution on mobile DSPs. The overall compilation time is also justified. In the future, we plan to design and integrate a more advanced (or customized) Quantization approach [42] to *GCD*^{2}, and explore DSP-friendly operator fusion [63] to further improve the performance.

Acknowledgment

The authors would like to thank the anonymous reviewers for their constructive comments and helpful suggestions. This work was supported in part by National Science Foundation (NSF) under the awards of CCF-2047516 (CAREER), CCF-2146873, CCF-2232813, CCF-2146852, CCF-2131509, CCF-2034850, and CCF-2007793, and Army Research Office/Army Research Laboratory via grant W911-NF-20-1-0167 to Northeastern University. Any errors and opinions are not those of the NSF, Army Research Office, or Department of Defense, and are attributable solely to the author(s).

References

[1] S. Vocke, H. Corporaal, R. Jordans, R. Corvino, and R. Nas, "Extending halide to improve software development for imaging dsps," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 3, pp. 1–25, 2017.

[2] C. Yang, S. Chen, Y. Wang, and J. Zhang, "The evaluation of dcnn on vector-simd dsp," *IEEE Access*, vol. 7, pp. 22 301–22 309, 2019.

[3] A. VanHattum, R. Nigam, V. T. Lee, J. Bornholt, and A. Sampson, "Vectorization for digital signal processors via equality saturation," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 874–886.

[4] M. B. S. Ahmad, A. J. Root, A. Adams, S. Kamil, and A. Cheung, "Vector instruction selection for digital signal processors using program synthesis," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 1004–1016. [Online]. Available: https://doi.org/10.1145/3503222.3507714

[5] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: An automated end-to-end optimizing compiler for deep learning," in *OSDI 2018*, 2018, pp. 578–594.

[6] S. G. Bhaskaracharya, J. Demouth, and V. Grover, "Automatic kernel generation for volta tensor cores," *arXiv preprint arXiv:2006.12645*, 2020.

[7] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," *arXiv preprint arXiv:1802.04730*, 2018.

[8] A. Venkat, T. Rusira, R. Barik, M. Hall, and L. Truong, "Swirl: High-performance many-core cpu code generation for deep neural networks," *The International Journal of High Performance Computing Applications*, vol. 33, no. 6, pp. 1275–1289, 2019.

[9] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, "Taso: optimizing deep learning computation with automatic generation of graph substitutions," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 47–62.

[10] Y. Ding, L. Zhu, Z. Jia, G. Pekhimenko, and S. Han, "Ios: Inter-operator scheduler for cnn acceleration," *Proceedings of Machine Learning and Systems*, vol. 3, 2021.

[11] P. Hill, A. Jain, M. Hill, B. Zamirai, C.-H. Hsu, M. A. Laurenzano, S. Mahlke, L. Tang, and J. Mars, "Deftnn: Addressing bottlenecks for dnn execution on gpus via synapse vector elimination and near-compute data fission," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 786–799.

[12] P. M. Phothilimthana, A. Sabne, N. Sarda, K. S. Murthy, Y. Zhou, C. Angermueller, M. Burrows, S. Roy, K. Mandke, R. Farahani *et al.*, "A flexible approach to autotuning multi-pass machine learning compilers," in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2021, pp. 1–16.

[13] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar, "Deepx: A software accelerator for low-power deep learning inference on mobile devices," in *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2016, pp. 1–12.

Authorized licensed use limited to: William & Mary. Downloaded on June 01,2023 at 02:21:04 UTC from IEEE Xplore. Restrictions apply.

[14] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *OSDI 2016*. USA: USENIX Association, 2016, pp. 265–283.

[15] X. Jiang, H. Wang, Y. Chen, Z. Wu, L. Wang, B. Zou, Y. Yang, Z. Cui, Y. Cai, T. Yu, C. Lyu, and Z. Wu, "Mnn: A universal and efficient inference engine," in *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze, Eds., 2020, vol. 2, pp. 1–13.

[16] M. Xu, M. Zhu, Y. Liu, F. X. Lin, and X. Liu, "Deepcache: Principled cache for mobile deep vision," in *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 129–144.

[17] L. N. Huynh, Y. Lee, and R. K. Balan, "Deepmon: Mobile gpu-based deep learning framework for continuous vision applications," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. ACM, 2017, pp. 82–95.

[18] S. Yao, S. Hu, Y. Zhao, A. Zhang, and T. Abdelzaher, "Deepsense: A unified deep learning framework for time-series mobile sensing data processing," in *Proceedings of the 26th International Conference on World Wide Web*, ser. WWW '17. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, 2017, p. 351–360.

[19] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy, "Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints," in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. ACM, 2016, pp. 123–136.

[20] S. Jiang, L. Ran, T. Cao, Y. Xu, and Y. Liu, "Profiling and optimizing deep learning inference on mobile gpus," in *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2020, pp. 75–81.

[21] M. Wang, S. Ding, T. Cao, Y. Liu, and F. Xu, "Asymo: scalable and efficient deep-learning inference on asymmetric mobile cpus," in *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, 2021, pp. 215–228.

[22] L. L. Zhang, S. Han, J. Wei, N. Zheng, T. Cao, Y. Yang, and Y. Liu, "nn-meter: towards accurate latency prediction of deep-learning model inference on diverse edge devices," in *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, 2021, pp. 81–93.

[23] P. Gibson, J. Cano, J. Turner, E. J. Crowley, M. O'Boyle, and A. Storkey, "Optimizing grouped convolutions on edge devices," in *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2020, pp. 189–196.

[24] N. D. Lane, P. Georgiev, and L. Qendro, "Deepear: robust smartphone audio sensing in unconstrained acoustic environments using deep learning," in *Proceedings of the 2015 ACM international joint conference on pervasive and ubiquitous computing*, 2015, pp. 283–294.

[25] K. Hegde, R. Agrawal, Y. Yao, and C. W. Fletcher, "Morph: Flexible acceleration for 3d cnn-based video understanding," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 933–946.

[26] S. Kaufman, P. Phothilimthana, Y. Zhou, C. Mendis, S. Roy, A. Sabne, and M. Burrows, "A learned performance model for tensor processing units," *Proceedings of Machine Learning and Systems*, vol. 3, 2021.

[27] J. Shen, Y. Huang, Z. Wang, Y. Qiao, M. Wen, and C. Zhang, "Towards a uniform template-based architecture for accelerating 2d and 3d cnns on fpga," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 97–106.

[28] J. Zhao, B. Li, W. Nie, Z. Geng, R. Zhang, X. Gao, B. Cheng, C. Wu, Y. Cheng, Z. Li, P. Di, K. Zhang, and X. Jin, "Akg: Automatic kernel generation for neural processing units using polyhedral transformations," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1233–1248. [Online]. Available: https://doi-org.proxy.wm.edu/10.1145/3453483.3454106

[29] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen, "Polyhedral-model guided loop-nest auto-vectorization," in *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2009, pp. 327–337.

[30] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan, "When polyhedral transformations meet simd code generation," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 127–138.

[31] Y. Chen, C. Mendis, M. Carbin, and S. Amarasinghe, "Vegen: a vectorizer generator for simd and beyond," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 902–914.

[32] E. Georganas, S. Avancha, K. Banerjee, D. Kalamkar, G. Henry, H. Pabst, and A. Heinecke, "Anatomy of high-performance deep learning convolutions on simd architectures," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 830–841.

[33] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *International Conference on Machine Learning*. PMLR, 2019, pp. 6105–6114.

[34] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

Authorized licensed use limited to: William & Mary. Downloaded on June 01,2023 at 02:21:04 UTC from IEEE Xplore. Restrictions apply.

[35] B. Yang, W. Luo, and R. Urtasun, "Pixor: Real-time 3d object detection from point clouds," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, 2018, pp. 7652–7660.

[36] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, "Unpaired image-to-image translation using cycle-consistent adversarial networks," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2223–2232.

[37] Qualcomm, "Snpe," 2017. [Online]. Available: https://developer.qualcomm.com/software/qualcomm-neural-processing-sdk

[38] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *PLDI 2013*. New York, NY, USA: Association for Computing Machinery, 2013, p. 519–530.

[39] Qualcomm, "Snapdragon 865," 2019. [Online]. Available: https://www.qualcomm.com/products/snapdragon-865-5g-mobile-platform

[40] ——, "Snapdragon 820," 2016. [Online]. Available: https://www.qualcomm.com/products/snapdragon-820-mobile-platform

[41] ——, "Theoretical speed of hexagon dsp," 2021. [Online]. Available: https://en.wikipedia.org/wiki/Qualcomm_Hexagon

[42] M. Cowan, T. Moreau, T. Chen, J. Bornholt, and L. Ceze, "Automatic generation of high-performance quantized machine learning kernels," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, 2020, pp. 305–316.

[43] H. Qin, Z. Cai, M. Zhang, Y. Ding, H. Zhao, S. Yi, X. Liu, and H. Su, "Bipointnet: Binary neural network for point clouds," *arXiv preprint arXiv:2010.05501*, 2020.

[44] Qualcomm, "Hexagon nn library," 2019. [Online]. Available: https://developer.qualcomm.com/software/hexagon-dsp-sdk

[45] R. Lee, S. I. Venieris, L. Dudziak, S. Bhattacharya, and N. D. Lane, "Mobisr: Efficient on-device super-resolution through heterogeneous mobile processors," in *The 25th Annual International Conference on Mobile Computing and Networking*, 2019, pp. 1–16.

[46] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[47] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 2704–2713.

[48] A. Jain, S. Bhattacharya, M. Masuda, V. Sharma, and Y. Wang, "Efficient execution of quantized deep learning models: A compiler approach," *arXiv preprint arXiv:2006.10226*, 2020.

[49] Q. Jin, L. Yang, and Z. Liao, "Adabits: Neural network quantization with adaptive bit-widths," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 2146–2156.

[50] S. Kim, A. Gholami, Z. Yao, M. W. Mahoney, and K. Keutzer, "I-bert: Integer-only bert quantization," *arXiv preprint arXiv:2101.01321*, 2021.

[51] H. Xie, Y. Song, L. Cai, and M. Li, "Overflow aware quantization: Accelerating neural network inference by low-bit multiply-accumulate operations." in *IJCAI*, 2020, pp. 868–875.

[52] A. VanHattum, R. Nigam, V. T. Lee, J. Bornholt, and A. Sampson, "A synthesis-aided compiler for dsp architectures (wip paper)," in *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 2020, pp. 131–135.

[53] X. Jiao, Y. Yin, L. Shang, X. Jiang, X. Chen, L. Li, F. Wang, and Q. Liu, "Tinybert: Distilling bert for natural language understanding," *arXiv preprint arXiv:1909.10351*, 2019.

[54] A. Anderson and D. Gregg, "Optimal dnn primitive selection with partitioned boolean quadratic programming," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, 2018, pp. 340–351.

[55] L. Hames and B. Scholz, "Nearly optimal register allocation with pbqp," in *Joint Modular Languages Conference*. Springer, 2006, pp. 346–361.

[56] C. Lee, J. K. Lee, T. Hwang, and S.-C. Tsai, "Compiler optimization on vliw instruction scheduling for low power," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 8, no. 2, pp. 252–268, 2003.

[57] G. Wang, W. Gong, and R. Kastner, "Instruction scheduling using max-min ant system optimization," in *Proceedings of the 15th ACM Great Lakes symposium on VLSI*, 2005, pp. 44–49.

[58] Qualcomm, "Hexagon v66 manual," 2017. [Online]. Available: QualcommHexagonV66Programmer'sReferenceManual

[59] C. Six, S. Boulmé, and D. Monniaux, "Certified and efficient instruction scheduling: application to interlocked vliw processors," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–29, 2020.

[60] D. Bernstein and M. Rodeh, "Global instruction scheduling for superscalar machines," in *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, 1991, pp. 241–255.

[61] R. C. Rocha, V. Porpodas, P. Petoumenos, L. F. Góes, Z. Wang, M. Cole, and H. Leather, "Vectorization-aware loop unrolling with seed forwarding," in *Proceedings of the 29th International Conference on Compiler Construction*, 2020, pp. 1–13.

Authorized licensed use limited to: William & Mary. Downloaded on June 01,2023 at 02:21:04 UTC from IEEE Xplore. Restrictions apply.

[62] W. Niu, X. Ma, S. Lin, S. Wang, X. Qian, X. Lin, Y. Wang, and B. Ren, "Patdnn: Achieving real-time dnn execution on mobile devices with pattern-based weight pruning," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 907–922.

[63] W. Niu, J. Guan, Y. Wang, G. Agrawal, and B. Ren, "Dnnfusion: accelerating deep neural networks execution with advanced operator fusion," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 883–898.

[64] H. Guan, S. Liu, X. Ma, W. Niu, B. Ren, X. Shen, Y. Wang, and P. Zhao, "Cocopie: enabling real-time ai on off-the-shelf mobile devices via compression-compilation co-design," *Communications of the ACM*, vol. 64, no. 6, pp. 62–68, 2021.

[65] LLVM, "Llvm," 2021. [Online]. Available: https://llvm.org/

[66] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan *et al.*, "Searching for mobilenetv3," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 1314–1324.

[67] J. Johnson, A. Alahi, and L. Fei-Fei, "Perceptual losses for real-time style transfer and super-resolution," in *European conference on computer vision*. Springer, 2016, pp. 694–711.

[68] J. Yu, Y. Fan, J. Yang, N. Xu, X. Wang, and T. S. Huang, "Wide activation for efficient and accurate image super-resolution," *arXiv preprint arXiv:1808.08718*, 2018.

[69] M. Tan, R. Pang, and Q. V. Le, "Efficientdet: Scalable and efficient object detection," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 10 781–10 790.

[70] A. Gulati, J. Qin, C.-C. Chiu, N. Parmar, Y. Zhang, J. Yu, W. Han, S. Wang, Z. Zhang, Y. Wu *et al.*, "Conformer: Convolution-augmented transformer for speech recognition," *arXiv preprint arXiv:2005.08100*, 2020.

[71] TensorFlow, "Post-training quantization," 2021. [Online]. Available: https://www.tensorflow.org/lite/performance/post_training_quantization

[72] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *CVPR 2009*, 2009, pp. 248–255.

[73] E. Agustsson and R. Timofte, "Ntire 2017 challenge on single image super-resolution: Dataset and study," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, July 2017.

[74] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *European conference on computer vision*. Springer, 2014, pp. 740–755.

[75] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, "Vision meets robotics: The kitti dataset," *International Journal of Robotics Research (IJRR)*, 2013.

[76] V. Panayotov, G. Chen, D. Povey, and S. Khudanpur, "Librispeech: An asr corpus based on public domain audio books," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2015, pp. 5206–5210.

[77] Qualcomm, "Snapdragon profiler," 2016. [Online]. Available: https://developer.qualcomm.com/software/snapdragon-profiler

[78] Google, "Snapdragon 820," 2020. [Online]. Available: https://coral.ai/docs/edgetpu/benchmarks/

[79] P. Torelli and M. Bangale, "Measuring inference performance of machine-learning frameworks on edge-class devices with the mlmark benchmark," *Techincal Report. Available online: https://www. eembc. org/techlit/articles/MLMARK-WHITEPAPERFINAL-1. pdf (accessed on 5 April 2021)*, 2021.

[80] L. Song, F. Chen, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "Accpar: Tensor partitioning for heterogeneous deep learning accelerators," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 342–355.

[81] W. Lin, C. G. Lee, and P. Chow, "an optimizing compiler for the tms320c25 dsp chip," in *Proc. Int. Conf. Signal Processing Applicat. Technol.*, no. 5. Citeseer, 1994, pp. I–689.

[82] H. De Man, J. Rabaey, P. Six, and L. Claesen, "Cathedral-ii: A silicon compiler for digital signal processing," *IEEE Design & Test of Computers*, vol. 3, no. 6, pp. 13–25, 1986.

[83] V. Zivojnovic, S. Pees, C. Schlager, M. Willems, R. Schoenen, and H. Meyr, "Dsp processor/compiler co-design: a quantitative approach," in *Proceedings of 9th International Symposium on Systems Synthesis*. IEEE, 1996, pp. 108–113.

[84] J. Xiong, J. Johnson, R. Johnson, and D. Padua, "Spl: A language and compiler for dsp algorithms," *ACM SIGPLAN Notices*, vol. 36, no. 5, pp. 298–308, 2001.

[85] M. Lorenz, P. Marwedel, T. Drager, G. Fettweis, and R. Leupers, "Compiler based exploration of dsp energy savings by simd operations," in *ASP-DAC 2004: Asia and South Pacific Design Automation Conference 2004 (IEEE Cat. No. 04EX753)*. IEEE, 2004, pp. 839–842.

[86] S. Rajagopalan, S. P. Rajan, S. Malik, S. Rigo, G. Araujo, and K. Takayama, "A retargetable vliw compiler framework for dsps with instruction-level parallelism," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 11, pp. 1319–1328, 2001.

[87] C.-K. Chen, L.-H. Tseng, S.-C. Chen, Y.-J. Lin, Y.-P. You, C.-H. Lu, and J.-K. Lee, "Enabling compiler flow for embedded vliw dsp processors with distributed register files," in *Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, 2007, pp. 146–148.

[88] D. Nuzman, I. Rosen, and A. Zaks, "Auto-vectorization of interleaved data for simd," *ACM SIGPLAN Notices*, vol. 41, no. 6, pp. 132–143, 2006.

Authorized licensed use limited to: William & Mary. Downloaded on June 01,2023 at 02:21:04 UTC from IEEE Xplore. Restrictions apply.

[89] G. Mainland, R. Leshchinskiy, and S. P. Jones, "Exploiting vector instructions with generalized stream fusion," *Communications of the ACM*, vol. 60, no. 5, pp. 83–91, 2017.

[90] D. G. Spampinato, D. Fabregat-Traver, P. Bientinesi, and M. Püschel, "Program generation for small-scale linear algebra applications," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, 2018, pp. 327–339.

[91] F. Franchetti, S. Kral, J. Lorenz, and C. W. Ueberhuber, "Efficient utilization of simd extensions," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 409–425, 2005.

[92] G. Ren, P. Wu, and D. Padua, "Optimizing data permutations for simd devices," *ACM SIGPLAN Notices*, vol. 41, no. 6, pp. 118–131, 2006.

[93] L. Chen, P. Jiang, and G. Agrawal, "Exploiting recent simd architectural advances for irregular applications," in *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2016, pp. 47–58.

[94] C. Mendis and S. Amarasinghe, "Goslp: Globally optimized superword level parallelism framework," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–28, 2018.

[95] J. Huh and J. Tuck, "Improving the effectiveness of searching for isomorphic chains in superword level parallelism," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2017, pp. 718–729.

[96] G. De Micheli, *Synthesis and optimization of digital circuits*. McGraw Hill, 1994, no. BOOK.

[97] R. Leupers, "Instruction scheduling for clustered vliw dsps," in *Proceedings 2000 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. PR00622)*. IEEE, 2000, pp. 291–300.

[98] J.-B. Tristan and X. Leroy, "Formal verification of translation validators: a case study on instruction scheduling optimizations," in *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2008, pp. 17–27.

[99] ——, "A simple, verified validator for software pipelining," *ACM Sigplan Notices*, vol. 45, no. 1, pp. 83–92, 2010.

[100] B. D. De Dinechin, "From machine scheduling to vliw instruction scheduling," *ST Journal of Research*, vol. 1, no. 2, pp. 1–35, 2004.

Authorized licensed use limited to: William & Mary. Downloaded on June 01,2023 at 02:21:04 UTC from IEEE Xplore. Restrictions apply.