Optimizing GPU Deep Learning Operators with Polyhedral Scheduling Constraint Injection

Cedric Bastoul*, Zhen Zhang*, Harenome Razanajato*, Nelson Lossing*, Adilla Susungi*,
Javier de Juan*, Etienne Filhol*, Baptiste Jarry*, Gianpietro Consolaro*, Renwei Zhang†
*Huawei Technologies France, Paris, France
†Huawei Technologies Co., Ltd., Beijing, China

2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) | 978-1-6654-0584-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/CGO53902.2022.9741260

***Abstract*—Automatic parallel code generation from high-level abstractions such as those manipulated by artificial intelligence and deep learning (AI/DL) frameworks heavily rely on compiler techniques for automatic parallelization and optimization. Many recent advances rely on the polyhedral framework for this task because of its ability to model and to apply a wide range of loop transformations. However, modeling the complexity of the target architecture and of efficient cost models to decide about the best transformation is in general out of reach for a framework based on linear/affine constraints. In this work, we propose to decouple the polyhedral framework into linear and non-linear components. We introduce the constraint tree abstraction which may be generated by a non-linear optimizer and injected to the polyhedral optimization process to build better solutions. We present how to benefit from such a mechanism to generate efficient codes for GPU in the context of AI/DL operators. Our constraint injection allows to drive the polyhedral scheduler towards efficient solutions for load/store vectorization relying both on memory coalescing and vector types. We implemented our scheduler supporting constraint injection and our constraint construction system within a production AI/DL framework. Experiments on well known neural networks show the efficiency of this approach with respect to state-of-the-art polyhedral scheduling for GPU.**

***Index Terms*—Polyhedral model, scheduling, vectorization**

I. Introduction

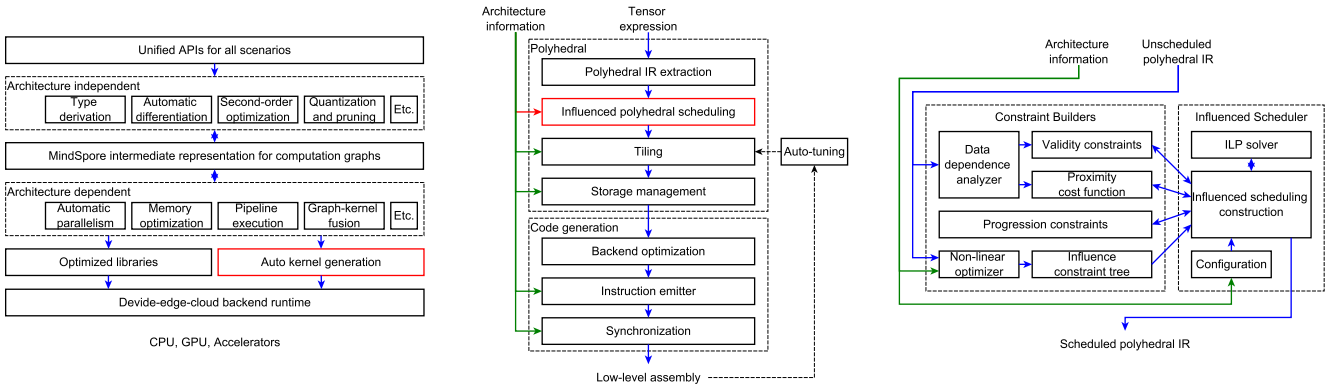
Automatic parallelization and optimization play a significant role in bridging the gap between high-level data/computation abstractions and their complex mapping to parallel CPUs/accelerators. Many advances in this field have been enabled using the polyhedral model, an algebraic representation of programs that builds on linear algebra to analyze and manipulate computational kernels [1]. Artificial intelligence and deep learning (AI/DL) frameworks mostly manipulate tensors and operators on those tensors that fit well the model. It resulted in an increasing interest for this approach in that context [2]–[6]. However, a limitation of the polyhedral model is the need to rely on affine constraints and cost functions to decide iteration-level scheduling, which may not be expressive enough to handle the complexity of the target architecture.

Iteration-level scheduling is responsible for a variety of critical optimization actions and decisions which may conflict with each other. It achieves parallelism extraction, exposing parallel blocks and loops, either coarse grain (external parallel loops), or fine grain (internal parallel/vector loops), or both. It performs permutability extraction, exposing loops that can be partitioned into smaller chunks with a subsequent tiling transformation. It takes the loop fusion/distribution decision, handles data locality optimization, enforces specific data access patterns to enable, e.g., vectorization, etc. Depending on the input problem and target architecture, some scheduling objectives may be more critical than others.

One of the polyhedral model's strength is to deeply integrate data dependencies, which allows to reason about a space of scheduling solutions that preserve the computation semantics. However the decision's complexity makes it difficult to choose the best solution relying only on models that translate well into linear optimization problems. Load/store vectorization on GPUs is an example of a key optimization which, in general, is hard to model using affine constraints only. Modeling decisions such as exploiting memory coalescing or using explicit vector types, choosing the number of array dimensions where consecutive accesses should be consecutive, or choosing the sequence of loop dimensions to carry vectorized accesses may be more appropriately handled by a non-linear approach (e.g., an algorithm with a non affine cost model). Unfortunately it would not fit the polyhedral model.

To solve this issue, we propose a general approach to let a non-linear optimizer influence the construction of a solution by a polyhedral scheduler thanks to constraint injection. We introduce the *influence constraint tree* an abstraction which specifies multiple prioritized optimization scenarios in the form of desirable affine constraints that may span over multiple scheduling dimensions. Influence constraint trees may be generated by a non-linear optimizer and processed by our polyhedral scheduler with an adapted algorithm, hence benefiting from non-linear guidance in addition to polyhedral model's semantics preservation and linear cost functions.

We present a non-linear optimizer to generate influence constraint trees that guide a polyhedral scheduler towards better optimizations for AI/DL fused operators on GPU. Specifically, the non-linear optimizer targets load/store vectorization which is not well addressed by existing polyhedral schedulers. Its strategy favors the usage of vector types over memory coalescing and prepares computation mapping and explicit vectorization by other compiler passes. We implemented both the polyhedral scheduler supporting constraint injection and the non-linear optimizer producing the constraints to be injected in a production AI/DL framework. Performance evaluation shows 1.7x geomean improvement over state-of-the-art scheduling for fused operators from various typical neural networks.



(a) MindSpore AI/DL Framework Overview (b) Auto Kernel Generator (AKG) (c) Influenced Polyhedral Scheduler

Fig. 1. Integration and Architecture of a Polyhedral Scheduler Supporting Constraint Injection in the AI/DL Framework MindSpore

II. Context and Problem Statement

AI/DL frameworks are complex software systems that take as input a high-level description of AI models, then generate code, deploy, manage and scale them for the target architecture. Without loss of generality, in this work we consider the Open Source project MindSpore [7], [8] as it is specifically designed to address multiple scenarios, from edge to cloud, and necessitates parallelization and optimization that should be easily retargeted according to the hardware architecture, e.g., CPU, GPU or NPU accelerators.

Figure 1 presents an overview of our work integration from the high-level vision of MindSpore, down to the architecture of our polyhedral scheduler supporting constraint injection. MindSpore (presented in Figure 1(a)) manipulates an intermediate representation of a computation graph and applies a series of processing to prepare efficient final code and runtime. This process includes the decision to apply AI/DL operator fusion ("Graph-kernel fusion" phase) to remove intermediate allocations, improve computation sharing and enable more optimization [9]–[11]. Code generation for fused operators is delegated to the Automatic Kernel Generator (AKG, shown in Figure 1(b)). AKG relies on the polyhedral compilation approach to perform fused operator parallelization, tiling and data management, then on lower-level code generation passes to handle backend optimizations [6].

A key process within polyhedral compilation frameworks is *scheduling*, which reorganizes statement executions to extract parallelism, achieve data locality and enable tiling. State-of-the-art polyhedral schedulers relying on pre-defined optimization strategies may not optimize some AI/DL operators in the best way for the desired target architecture. For instance, let us consider the computational kernel shown in Figure 2(a). It is a simplified version of a real-life fused operator submitted to AKG that we will use as a running example throughout this document. State-of-the-art polyhedral scheduler such as isl scheduler [12], [13] (used natively in AKG) can analyze the kernel and automatically parallelize and optimize it up to the final version shown in Figure 2(b). While the polyhedral scheduler successfully extracted parallel loops denoted by "forall" keywords, the final code is far from optimal when targeting GPU or AI/DL accelerators because (1) it is not a perfectly nested loop and (2) the access to the main tensor `D` is inefficient due to long jumps in the memory space at every iteration of the innermost loop. The present work aims at enabling fine control over the polyhedral scheduler to influence it towards better solutions, e.g., to solve identified issues when optimizing codes like in Figure 2.

```
for (i = 0; i < N; i++)
  for (k = 0; k < N; k++)
X:  B[i][k] = f(A[i][k]);

for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < N; k++)
Y:    C[i][j] = g(C[i][j], B[i][k], D[k][i][j]);
```

(a) Initial pseudo-code corresponding to the fused operator before polyhedral scheduling for parallelism extraction and optimization

```
forall (i = 0; i < N; i++)
  forall (k = 0; k < N; k++)
X:  B[i][k] = f(A[i][k]);

forall (i = 0; i < N; i++)
  forall (j = 0; j < N; j++)
    for (k = 0; k < N; k++)
Y:    C[i][j] = g(C[i][j], B[i][k], D[k][i][j]);
```

(b) Typical optimized version of the pseudo-code using polyhedral scheduling, e.g., isl (version 0.22, default options) or Pluto (version 0.11.4, options `--parallel --nointratileopt` to focus on initial scheduling), the outermost loops marked with the forall keyword have been parallelized, loops are split and the memory access of tensor D is inefficient

```
forall (i = 0; i < N; i++)
  for (k = 0; k < N; k++) {
X:  B[i][k] = f(A[i][k]);
    forvec (j = 0; j < N; j++)
Y:    C[i][j] = g(C[i][j], B[i][k], D[k][i][j]);
  }
```

(c) Desired optimized version of the pseudo-code with external parallelism marked with the forall keyword, inner vectorizable loop marked with the forvec keyword, and efficient memory access patterns

Fig. 2. Running Example: simplified version (with abstracted computation and lower tensor dimensionality, but same structure and issues for example purpose) of a real fused operator from BERT natural language processing network (named `fused_mul_sub_mul_tensoradd` in MindSpore/AKG), both arrays `B` and `C` hold output values

III. Background and Notations

Polyhedral schedulers are using linear algebra to model and to compute an ordering for all iterations of a computational kernel. This ordering is expressed, for each statement, in the form of a multidimensional affine function which associates iterations of the statement to a logical date. For a better understanding of the present work, it is necessary to understand the two main abstractions manipulated by polyhedral schedulers and their notations: iteration domains which represent executions of statements, and affine scheduling functions which encode the ordering. In the following, $\vec{v}$ denotes a column vector while bold $\boldsymbol{v}$ denotes a row vector. To simplify notations when writing column vectors in-line with other text, we omit the transpose operator and use commas to separate column vector elements: $\binom{a}{b}$ is equivalent to (a, b). The row vector corresponding to the n^{th} row of a matrix M is noted $M_{n,\bullet}$.

A. Iteration Domains

The application domain of polyhedral schedulers is loop-based programs where the bounds of the loops and the conditions of the tests are affine constraints on the loop iterators and the global parameters (i.e., numbers that are not known but have fixed value during the execution of the computational kernel). The vector of all global parameters is noted $\vec{p}$, e.g., for the program in Figure 2(a) we have $\vec{p} = (N)$.

In the target class of programs, a particular execution of a statement can be totally determined by the value of the surrounding loop iterators, called the *iteration vector* and noted $\vec{\imath}$, e.g., for the program in Figure 2(a), the execution of statement X for $i = 1$ and $k = 2$ corresponds to the iteration vector $\vec{\imath}_X = (1, 2)$ and we note that execution $X(1, 2)$. We can represent all the executions of a statement S of the computational kernel with its set of all possible iteration vectors $\mathcal{D}_S(\vec{p}) = \{\vec{\imath}_S\}$.

For instance let us consider the computational kernel shown in Figure 2(a). The computational kernel has two statements X and Y. Statement X is enclosed inside two loops and any execution of the statement X is determined by the iteration vector (i, k). Hence, all the executions of X can be modeled by the set of all possible iteration vectors:

$$\mathcal{D}_X(N) = \left\{ \begin{pmatrix} i \\ k \end{pmatrix} \middle| \; 0 \le i < N, 0 \le k < N \right\},$$

Equivalently for statement Y we have:

$$\mathcal{D}_Y(N) = \left\{ \begin{pmatrix} i \\ j \\ k \end{pmatrix} \middle| \; 0 \le i < N, 0 \le j < N, 0 \le k < N \right\},$$

B. Affine Scheduling Functions

Affine scheduling functions aim at specifying the relative ordering of all iterations of all iteration domains (note that iteration domains do not encode ordering, they are only sets in the mathematical sense). To model such ordering, polyhedral schedulers use multidimensional affine functions that associate every iteration of iteration domains to a logical date. Each dimension of the function is an affine expression of the iteration vector dimensions and the parameters. Logical dates are multidimensional: they encode a date with several components in a lexicographic way (like days, hours, minutes, seconds, etc.). It was soon shown that such affine scheduling functions are expressive enough to model arbitrary sequences of all classical loop transformations (loop fusion, fission, reversal, interchange, skewing, strip-mining, tiling, shifting, etc.) [14]–[16], hence the power of such abstraction.

For instance, the following scheduling functions are a (non unique) way to encode the order of the iterations in the example computational kernel in Figure 2(a):

$$\theta_X \begin{pmatrix} i \\ k \\ N \end{pmatrix} = \begin{pmatrix} 0 \\ i \\ k \\ 0 \end{pmatrix}, \; \theta_Y \begin{pmatrix} i \\ j \\ k \\ N \end{pmatrix} = \begin{pmatrix} 1 \\ i \\ j \\ k \end{pmatrix}.$$

In this example we have one scheduling function for each statement, both are 4-dimensional and map iterations to the same target time space. For instance it specifies that the iteration $i = 2$ and $k = 1$ of statement X, noted $X(2, 1)$ is executed at logical date $\theta_X(2, 1, N) = (0, 2, 1, 0)$, in other words at "day 0, hour 2, minute 1, second 0". Similarly it states that the iteration $i = 2, j = 0, k = 1$ of statement Y, noted $Y(2, 0, 1)$, is executed at logical date $\theta_Y(2, 0, 1, N) = (1, 2, 0, 1)$, in other words at "day 1, hour 2, minute 0, second 1". Hence $X(2, 1)$ is executed before $Y(2, 0, 1)$. We may note all iterations of X are executed at "day 0" while all iterations of Y are executed at "day 1", which models the separation of the two external loops in the computational kernel. The second dimension of X corresponds to expression i (specifying that lower values of i are executed before higher values of i) which corresponds to the first loop, and the same reasoning applies to all other scheduling dimensions. Finally we may check that the scheduling functions model a total order for all iterations and that it corresponds to the iteration order in the example computational kernel.

There is no limit to the number of scheduling dimensions, but the expression for each dimension must be an affine expression of the iteration vector dimensions and the parameters. Under this assumption, we may rely on existing algorithms and tools to generate a code that implements any ordering modeled in this way [17]–[19]. Hence the general form of scheduling functions for a statement S is $\theta_S(\vec{\imath}_S, \vec{p}) = T_S(\vec{\imath}_S, \vec{p}, 1)$, where T_S is the *transformation matrix* for S. E.g., for our example the scheduling function for statement X is:

$$\theta_X \begin{pmatrix} i \\ k \\ N \end{pmatrix} = T_X \begin{pmatrix} i \\ k \\ N \\ 1 \end{pmatrix} = \begin{bmatrix} T_{X,0,0} & T_{X,0,1} & T_{X,0,2} & T_{X,0,3} \\ T_{X,1,0} & T_{X,1,1} & T_{X,1,2} & T_{X,1,3} \\ \cdots & & & \\ T_{X,n,0} & T_{X,n,1} & T_{X,n,2} & T_{X,n,3} \end{bmatrix} \begin{pmatrix} i \\ k \\ N \\ 1 \end{pmatrix}.$$

In the particular case of one-dimensional scheduling functions, we use the notation ϕ instead of θ: the d^{th} dimension of the scheduling function for statement S is $\phi_{S,d}(\vec{\imath}_S, \vec{p}) = T_{S,d,\bullet}(\vec{\imath}_S, \vec{p}, 1)$. E.g., for our example, dimension 1 of the scheduling function for statement X is:

$$\phi_{X,1} \begin{pmatrix} i \\ k \\ N \end{pmatrix} = T_{X,1,\bullet} \begin{pmatrix} i \\ k \\ N \\ 1 \end{pmatrix} = \begin{bmatrix} T_{X,1,0} & T_{X,1,1} & T_{X,1,2} & T_{X,1,3} \end{bmatrix} \begin{pmatrix} i \\ k \\ N \\ 1 \end{pmatrix}.$$

The role of the polyhedral scheduler is to compute scheduling functions, which corresponds to finding the various scheduling

coefficients of the transformation matrix for all statements (e.g., $T_{X,0,0}$ denoting the scheduling coefficient multiplying i at dimension 0 for statement X). State-of-the-art scheduling algorithms are able to compute the coefficients in a systematic way. The present work aims at computing them in a controlled way to achieve better optimizations.

IV. Scheduling with Constraint Injection

The automatic construction of optimizing affine scheduling functions is typically an iterative process where scheduling dimensions are computed one after the other, from the outermost to the innermost, until we get enough dimensions. At each step, we compute the scheduling coefficients of a given dimension for all statements. The computation itself is an integer linear optimization problem built from a variety of linear constraints and objective functions depending on the desired optimization properties. This process derives from the seminal work of Feautrier [20], [21] which has been the basis for many subsequent advances proposing adaptations and/or extensions to achieve specific goals [12], [22]–[25].

Differently from related work, our construction algorithm decouples constraint construction from constraint expression to enable non-linear optimization strategies in a linear scheduling framework. Furthermore, while backtracking to come back to outermost dimension construction is only a particular case in existing frameworks (e.g., isl scheduling may come back to the computation of the immediate previous scheduling dimension to replace the Pluto strategy with the Feautrier strategy [12]), our work deeply relies on backtracking to any dimension and inter-dimension constraints in its construction mechanism.

To provide more control over our scheduling construction algorithm, we distinguish different sets of constraints (which restrict the space of solutions of the linear problem) and cost functions (which provide guidance to decide about the best solution within the space), and we assign them a priority that will affect the algorithm backtracking process. They are generated by specific constraint builders that are detailed in Section IV-A. In particular, we present our specific *influence constraint tree* abstraction, designed to influence the scheduling construction process in multiple linked dimensions. Then we present our influenced scheduling construction algorithm in Section IV-B.

A. Constraint Builders

1) Validity Constraints: The most important property of a scheduling function is to respect the original program semantics. *Validity constraints* restrict the solution space to only such scheduling functions. Their construction has been introduced by Feautrier [20]. It derives from the fact that a sufficient constraint for validity is the preservation of the original relative ordering of any pair of statement executions that depend on each other. E.g., if a "*source*" statement iteration (for instance, iteration $X(1,2)$ in Figure 2(a)) produces a data which is used by another "*target*" statement iteration (for instance, iteration $Y(1,0,2)$), then the logical date of the source iteration must be lower than that of the target iteration.

We use the dependence relation abstraction (also called *dependence between iterations*) to model sets of pairs of statements in dependence relation in the most precise way [20], [26]. We note such set of pairs of iterations $\vec{t}$ from a target statement T that depend on iterations $\vec{s}$ from a source statement S: $\delta_{S\to T}(\vec{p}) = \{\langle \vec{s}, \vec{t} \rangle\}$. The dependence relation captures the constraints met by statement executions when they are dependent. First, source and target statement executions must actually exist, i.e., they have to belong to their corresponding iteration domains. Then, they must access the same memory cell, one of these accesses being a write. Hence they must access the same scalar or array, and in the later case, array indices must be equal. Finally, the source must be executed before the target in the original code, i.e. when considering the initial statement execution ordering of the input code, the source execution order must be lower than that of the target. In our application domain, all those constraints translate to affine (in)equalities, and all dependencies can be captured by a set of dependence relations [20]. E.g., in Figure 2(a)) the dependence relation modeling the flow dependencies (read-after-write) between source iterations of statement X and target iterations of statement Y accessing the same array B is:

$$\delta_{X\to Y}(N) = \left\{ \left\langle \begin{pmatrix} i_X \\ k_X \end{pmatrix}, \begin{pmatrix} i_Y \\ j_Y \\ k_Y \end{pmatrix} \right\rangle \;\middle|\; \begin{array}{rcl} 0 \leq & i_X & < N \\ 0 \leq & k_X & < N \\ 0 \leq & i_Y & < N \\ 0 \leq & j_Y & < N \\ 0 \leq & k_Y & < N \\ i_X & = & i_Y \\ k_X & = & k_Y \end{array} \right\}.$$

It follows, for a one-dimensional scheduling to be valid, that the validity constraint is:

$$\forall S,\ \forall T,\ \forall \langle \vec{s}, \vec{t} \rangle \in \delta_{S\to T}(\vec{p}),\ \phi_S(\vec{s}, \vec{p}) \leq \phi_T(\vec{t}, \vec{p}). \quad (1)$$

Intuitively it means that the source iteration has to be scheduled before (or at the same time as) the target iteration. When we have $\phi_S(\vec{s}, \vec{p}) < \phi_T(\vec{t}, \vec{p})$, we say that the dependence relation is *strongly satisfied*. When we have $\phi_S(\vec{s}, \vec{p}) \leq \phi_T(\vec{t}, \vec{p})$ (i.e., the logical dates are the same for at least one $\langle \vec{s}, \vec{t} \rangle$), we say that the dependence relation is *weakly satisfied*. Multi-dimensional scheduling may be necessary to strongly satisfy all dependence relations at most at the last dimension.

The validity constraint as formulated in equation 1 is not affine because unknowns (elements of the transformation matrix T_S) multiply variables (elements of the vector $(\vec{\imath}_S, \vec{p}, 1)$). However Feautrier showed how to rewrite it as a set of affine constraints thanks to the affine form of Farkas' lemma [20]. For space reason we do not detail the process here and point to reader, e.g., to Bondhugula's PhD document for detailed examples [27]. Every dependence relation may contribute a set of validity constraints. The validity constraint builder aims at providing them for a set of dependence relations.

2) Proximity Cost Functions: While validity constraints are fundamental to guarantee the scheduling correctness, other constraints and cost functions are necessary to choose good optimizing scheduling amongst the space of all legal solutions. Proximity cost model has been introduced by Bondhugula

et al. to extract coarse-grain parallelism and maximize data locality [23]. It aims at minimizing the data reuse distance, i.e., the difference between logical dates of two statement executions accessing the same memory cell: for every $\langle \vec{s}, \vec{t} \rangle \in \delta_{S \to T}$, where we may also consider input (read-after-read) dependencies, the reuse distance is the quantity $\phi_T(\vec{t}, \vec{p}) - \phi_S(\vec{s}, \vec{p})$.

To express the reuse distance minimization objective, we first define a bound of the reuse distance as an affine function of the global parameters: $\boldsymbol{u}.\vec{p} + w$. Then, we rely on (possibly lexicographic) linear optimization to minimize the coefficients (elements of $\boldsymbol{u}$ and w) of this affine function. We note the lexicographic minimization operator $\text{minimize}_{\prec}$. The ability to define an affine bound is directly related to the application domain where iterators are themselves bounded with affine functions of the parameters. The general expression of the proximity cost function is:

$$\forall S,\ \forall T,\ \forall \langle \vec{s}, \vec{t} \rangle \in \delta_{S \to T}(\vec{p}),\ \phi_T(\vec{s}, \vec{p}) - \phi_S(\vec{t}, \vec{p}) \leq \boldsymbol{u}.\vec{p} + w$$
$$\text{minimize}_{\prec}\ f_{\text{proximity}}(\vec{u}, w) \tag{2}$$

Where $f_{\text{proximity}}(\vec{u}, w)$ is a possibly multidimensional vector of linear expressions on the elements of $\boldsymbol{u}$ and w. Let us note u_i the i^{th} element of $\boldsymbol{u}$. The original work describing proximity cost function suggests $f_{\text{proximity}}(\vec{u}, w) = (u_0, u_1, \ldots, u_{\dim(\boldsymbol{u})-1}, w)$ [23] while implementation in isl library proposes[1] $f_{\text{proximity}}(\vec{u}, w) = \left(\sum_{i=0}^{\dim(\boldsymbol{u})-1} u_i, w\right)$ [13]. As our reference in this work is isl implementation, our work uses the later form of cost function.

An extreme way to force the reuse distance to be minimal is to force it to be zero. In this case the constraint is simply $\phi_S(\vec{s}, \vec{p}) = \phi_T(\vec{t}, \vec{p})$. Lim and Lam introduced this constraint, named *space-partition constraint* (a.k.a. *coincidence constraint* in isl scheduling implementation [13]), to extract synchronization-free parallelism [28]. Such constraint severely restrict the space of legal solutions and may not be satisfied, requiring to backtrack to a less constrained problem.

In the same way as validity constraints, the formulation of the bound on reuse distance is not an affine expression, but it can be rewritten using affine constraints by exploiting the affine form of Farkas' lemma. Hence the proximity cost function builder aims at producing the affine constraints to express the reuse distance bounds for a set of dependence relations and those forming the cost function to minimize.

3) Progression Constraints: A scheduling is complete when it defines a total order on all statement executions that are dependent on each other. In other words, all dependencies should be strongly satisfied by the final multidimensional scheduling: for each dependence relation, there must exist a scheduling dimension d that strongly satisfies the dependence relation while preceding dimensions only weakly satisfies it.

To ensure the scheduling construction is a finite process, every scheduling dimension should contribute to the construction of the total order. Non-contributing scheduling dimensions may either be "trivial" ones where all iterations are scheduled at the same date, or dimensions that are linearly dependent on preceding dimensions because they would specify a redundant ordering with respect to preceding dimensions. In particular, the *zero* solution where all scheduling coefficients are zero is a trivial solution to the validity constraint (1) which achieves the ideal proximity cost (2). Avoiding the zero solution and linearly dependent dimensions guarantees progression.

Several solutions have been suggested in previous work to ensure progression depending on whether the scheduling coefficients are all positive, bounded or unrestricted.

When we restrict scheduling coefficients to be positive, we may miss scheduling solutions that would correspond to code transformations involving loop reversal and loop skewing with negative factors. Such transformations may be useful in some cases, e.g., to optimize inter-nest locality [29] or to optimize stencil computation on periodic domains [24]. However, missing them is not detrimental to performance in general, as shown by the effectiveness of the Pluto algorithm that makes this assumption [23]. Considering only positive scheduling coefficients greatly simplifies progression constraint construction. In this case, requiring the sum of the iteration vector scheduling coefficients to be greater or equal to 1 is a sufficient condition. Hence, when constructing the d^{th} scheduling dimension, we should have [23]:

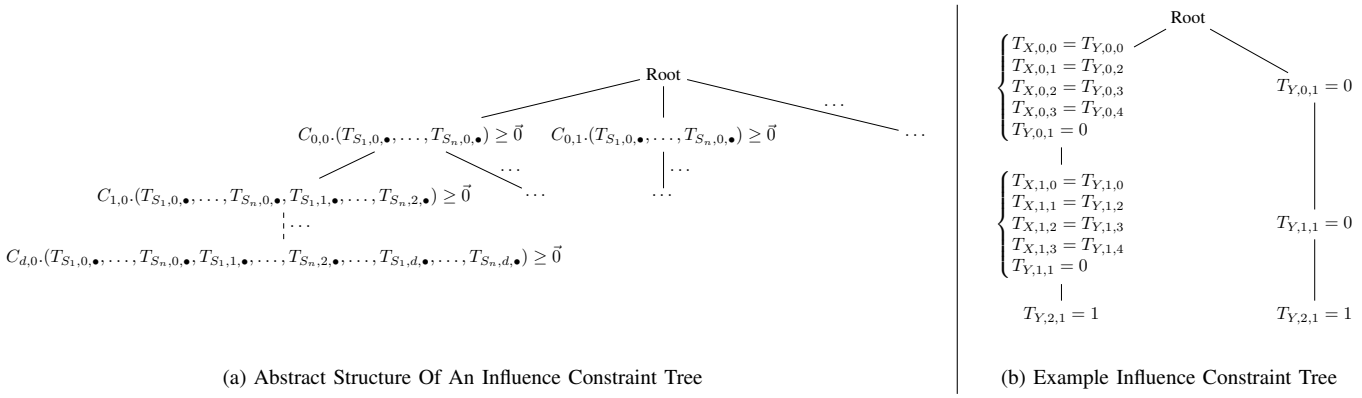
$$\forall S,\quad \sum_{i=0}^{\dim(\vec{\imath}_S)-1} T_{S,d,i} \geq 1 \tag{3}$$

To guarantee linear independence of the d^{th} scheduling dimension with respect to preceding dimensions, we may compute a basis of the subspace orthogonal to those preceding dimensions and derive constraints that the new dimension must respect to belong to that subspace [23]. Let us consider the already computed first dimensions of the scheduling function $\theta_S(\vec{\imath}_S, \vec{p})$ and decompose it in the following way: $\theta_S(\vec{\imath}_S, \vec{p}) = H_S \vec{\imath}_S + G_S \vec{p} + \vec{f}_S$. We note $H^{\perp}$ the matrix where each row corresponds to an component of the basis orthogonal to the row vectors of H_S. Various ways exist to construct $H^{\perp}$, e.g., Pluto algorithm uses $H_S^{\perp} = I - H_S^T (H_S H_S^T)^{-1} H_S$ [23], [30], while isl scheduling relies on the decomposition on Hermite normal form [13], [31]. For a given vector $\vec{h}_S$, we have $H_S^{\perp}.\vec{h}_S = \vec{0}$ iff either $\vec{h}_S$ is linearly dependent on row vectors of H_S or $\vec{h}_S = \vec{0}$. Hence making sure one component of $H_S^{\perp}.\vec{h}_S$ is not zero and $\vec{h}_S$ is not the zero vector guarantees linear independence. Pluto's practical solution is to consider the subspace where all constraints are non-negative [23]:

$$\forall S,\ \left(\forall d\ H_{S,d,\bullet}^{\perp}.\vec{h}_S \geq 0\right) \wedge \left(\sum_d H_{S,d,\bullet}^{\perp}.\vec{h}_S \geq 1\right) \tag{4}$$

Equations (3) and (4) provide sufficient constraints to guarantee progression. However they both over-constrain the problem and may disable profitable solutions, e.g., including negative scheduling coefficients. While there exist solutions to address negative coefficients [13], [24], in our AI/DL context we did not observe any case where negative coefficients were

[1]We omit here isl's decomposition of u_i into positive and negative parts.



(a) Abstract Structure Of An Influence Constraint Tree

(b) Example Influence Constraint Tree

Fig. 3. Influence Constraint Tree General Form And Example: example may influence scheduling of program in Figure 2. $T_{S,d,c}$ is the c^{th} scheduling coefficient at dimension d for statement S. Scheduling coefficients are associated to $(i_X, k_X, N, 1)$ for statement X and to $(i_Y, j_Y, k_Y, N, 1)$ for Y

computed by isl, and equations (3) and (4) are adequate. Hence the role of the progression constraint builder is to provide those constraints thanks to (e.g.,) equations (3) and (4) according to the already computed scheduling dimensions.

4) Influence Constraint Trees: Our work acknowledges the fact that polyhedral scheduling is expressive enough to model and to apply sufficiently complex loop transformations, but that not any objective function may be modeled using an affine representation. Instead, we provide a way to inject constraints to the scheduling algorithm with enough flexibility to handle many cases: *influence constraint trees*. The goal of influence constraint trees is to provide multiple optimization scenarios built by a non-linear optimizer with better vision about the optimization goals but no, or incomplete, vision about data dependencies and how to construct a complete scheduling that respects the original program semantics. This approach is transformation-centric: the non-linear optimizer will provide the linear polyhedral scheduler with desirable properties of the transformation to influence the construction of the final scheduling. Then the scheduling algorithm will build correct scheduling functions that respect the most profitable scenario according to the non-linear optimizer. If for some reason no optimization scenario is feasible, the scheduler output will be no different than a usual polyhedral scheduler.

An influence constraint tree specifies inter-statement and inter-dimension prioritized constraints on multidimensional scheduling function coefficients using an ordered tree abstraction. Each node at the d^{th} depth may contain a set of linear constraints on the scheduling coefficients of all statements from the first scheduling dimension to the d^{th}. Edges from one node at depth d to nodes at depth $d+1$ model different constraint alternatives. The ordering between siblings is significant and models the priority between alternatives. Figure 3(a) presents an abstract structure of an influence constraint tree for a program with n statements from S_1 to S_n. In each node, a matrix $C_{d,p}$, where d corresponds to the depth and p to the priority at that depth, stores the constraint information. Priorities are unique at a given depth and simply correspond to the left-to-right node ordering at a given depth of the ordered tree. Each row vector of C matrices corresponds to an affine inequality. Our implementation also supports the specification of new objective functions in each node (introducing new variables corresponding to each additional optimization objective and priorities to weight/order them in the possibly lexicographic optimization process), however it is not used in the constraint tree construction presented in Section V, hence we do not discuss them further.

The purpose of constraints specified in the influence constraint tree is to be injected during the scheduling construction process to influence it at a given depth. Depth of the tree corresponds in general to the construction of the corresponding scheduling dimension. As some constraints may be too restrictive for the construction to succeed, the structure of the tree proposes alternatives to be visited according to depth-first-search traversal. Once the construction successfully reaches a leaf, the contribution of the influence constraint tree terminates. Section IV-B describes the scheduling construction algorithm integrating constraint tree and appropriate backtracking mechanism.

While the constraint tree holds affine constraints, it may not be generated using an affine approach and its construction is prepared outside the affine scheduling algorithm. E.g., Section V presents such non-linear optimizer to improve mapping of AI/DL fused operators with efficient load/store vectorization on GPUs. Intuitively, the non-linear optimizer suggests desirable scheduling properties based on its own decision mechanism and lets the affine scheduler build the scheduling according to both those constraints and internal (validity, proximity, etc.) affine constraints.

For instance, an optimizer may analyze the code in Figure 2(a) and decide that (1) because of the data reuse on reference B, iterations accessing the same location should be scheduled together, (2) achieving vectorization on statement Y is important and (3) that vectorization is the most important optimization objective with respect to the architecture target. While the data reuse may be easily modeled with affine constraints such as proximity, vectorization requires non-affine decision in general (e.g., to address contiguous access on multiple array dimensions with understanding of the array size

and memory layout) and inter-scheduling-dimension constraints (e.g., to guarantee a given number of innermost loops access same or contiguous data). Furthermore, a mechanism to specify priorities is required.

Figure 3(b) shows an example of influence constraint tree to influence scheduling construction towards such requirements. The first branch suggests to schedule the two statements in the same way in the first two dimensions to influence data reuse, and to remain independent on j_K on the first two dimensions while having exactly a coefficient 1 for j_K on the third dimension to prepare vectorization. A second branch only keeps constraints related to vectorization to leave more freedom to the scheduling algorithm in case the first branch in not feasible. Ultimately the scheduler can build a solution while respecting constraints of the first branch, leading to final code in Figure 2(c).

Each constraint set may be augmented with constraints on meta-information such as considering the computation of a dimension is successful only if it is parallel, permutable, etc. Hence the role of the influence constraint tree builder is to build the constraint tree abstraction based on possibly non-linear optimization according to knowledge on both the input problem and the target architecture.

B. Influenced Scheduling Construction

Algorithm 1 presents our influenced scheduling construction. It is a modified version of the Pluto algorithm [23] with the integration of the support for influence constraint tree and with more explicit contributions of the various constraint builders to support different backtracking scenarios. Pluto algorithm computes a scheduling iteratively on each dimension, from the outermost to the innermost. Its goal is to build outermost sequences of parallel and permutable dimensions while improving data locality (permutability aiming at enabling subsequent tiling transformation). Intuitively a parallel dimension is found if for that dimension the reuse distance is zero (relates to the proximity cost function), a sequence of permutable dimensions is found if they satisfy the same set of dependence relations (relates to the validity constraints), and data locality is optimized if the reuse distance is minimized (again relating to the proximity cost function). The algorithm is pushed towards termination thanks to the progression constraints.

The heart of the construction system is solving integer linear programming problems composed with adequate constraints. We rely on lexicographic optimization offered by the isl library to address multiple objectives, configured in the same way as the isl scheduler to allow comparisons with the only addition of the influence constraints and processing [13]. Note that isl scheduler works in a different way than Pluto's: if it fails to find a solution with zero reuse distance, it changes the scheduling strategy for that dimension to Feautrier's [20], [21] to generate a sequential dimension strongly satisfying as many dependencies as possible and let more chance to find zero reuse distance solution at the next dimension [12]. We may use this mechanism as well but it was not necessary in the context of our experimental study, as fused AI/DL operators offer enough parallelism, hence we do not expose it.

If the ILP solver cannot find a solution for the desired dimension, a fallback mechanism is triggered to look for less desirable but semantically correct solution. We have implemented a customizable constraint de-activation process which turns off the lowest priority constraint set and restarts the solver, in the hope of finding a solution. The structure of the influence constraint tree holds priorities by design. Furthermore each contribution to the constraint system is assigned a priority. Algorithm 1 shows the configuration used in this work: when no solution is found, first we check whether influence is asking for a supplementary dimension and in this case we remove the

Algorithm 1: Influenced Scheduling Construction

Data: Dependence relations D, constraint tree C
Result: Transformation matrices $T_S\ \forall S$

1 d := 0;
2 node := first branch of C root node;
3 Initialize $T_{S,0,\bullet}$ to 0 row-vector $\forall S$;
4 **repeat**
5 $Backup[d]$:= D;
6 P := progression constraints for T_S;
7 V := validity constraints for D;
8 R := reuse distance bounding constraints for D;
9 I := influence constraints related to node;
10 $T_{S,d,\bullet}$:= solution to constraint system $P \wedge V \wedge R \wedge I$ optimizing proximity function;
11 **if** *No solution was found* **then**
12 **if** $D = \emptyset$ **then**
13 P := $\emptyset$;
14 **goto** line 7;
15 **end**
16 **if** $\exists$ *a right sibling to node* **then**
17 node := right sibling to node;
18 D := $Backup[d]$;
19 **goto** line 9;
20 **end**
21 **if** $\exists$ *an elt of* D *strongly satisfied by* T_S **then**
22 D := elts of D not strongly satisfied by T_S;
23 **goto** line 7;
24 **end**
25 **if** $\exists$ *a right sibling to ancestor* **then**
26 node := closest right sibling to ancestor;
27 d := depth of node;
28 D := $Backup[d]$;
29 withdraw dimensions of T_S that are $\geq d$;
30 **goto** line 6;
31 **end**
32 Select two or more strongly connected components in the dependence graph formed with elements of D and order them by inserting scalar dimensions in T_S;
33 D := elts of D not strongly satisfied by T_S;
34 **else**
35 Append the solution as a new dimension of T_S;
36 **end**
37 d := $d + 1$;
38 node := node left child if it exists, `NULL` otherwise;
39 **until** $D = \emptyset$ *and node is a leaf*;

progression constraints (line 12), second we try to integrate less desirable influence constraint for the same dimension (line 16), third we try to discard permutability property (line 21), fourth we backtrack to a previous dimension (line 25), fifth we separate strongly connected components (line 32). Ultimately if no influence scenario is feasible, the algorithm runs without any influence constraint.

Our design choice in this work is to rely on injected constraints rather than objective functions even though our implementation enables it. Objective functions are softer as they do not restrict the solution space. However as our purpose is to cover multidimensional scenarios (e.g., if a constraint is not met at a given depth d, some constraints at further depths may not be relevant anymore, as constraints related to vectorization in Figure 3(b)) they would necessitate a more complex fallback mechanism. Moreover, they require more variables in the ILP problem which may challenge scalability. Finally, in the context of AI/DL fused operators with usually limited/simple data dependencies we could observe only few activation of the backtracking, validating the design.

V. Application to GPU AI/DL Operators

Many AI/DL operators such as element-wise operators, have low computational intensity but may process large tensors, hence memory bandwidth is often a limiting factor in that context. Load/store vectorization is a key optimization on GPU to minimize the number of memory transactions and to increase the bandwidth utilization. It can be achieved through two main mechanisms on CUDA GPU architectures: memory coalescing and the usage of vector types. On one hand, memory coalescing combines simultaneous accesses to adjacent memory locations by adjacent threads to reduce the number of memory transaction. Indeed, threads are processed per groups of 32 called warps. A warp is the basic unit of execution in a GPU, hence the presence of a warp scheduler instead of what could seemingly be a thread scheduler in a streaming multiprocessor. Proper grouping and alignment of data directly impacts the number of transactions required to transfer data at the warp level. On the other hand, as all registers on a GPU are four-vectors, it is more advantageous to use vector types to load and store data through blocks of 64 or 128 bits. Both mechanisms are crucial to reduce latency and improve bandwidth utilization.

To this end, we automatically build influence constraint trees discussed in Section IV with different goals: (i) exhibit an innermost loop dimension purposefully prepared to be rewritten using explicit vector types by a backend pass, (ii) arrange enough following dimensions to maximize coalescing and (iii) enable freedom of scheduling choices on remaining dimensions. Hence we are looking for the shortest ordered list of innermost dimensions that minimizes the number of memory transactions. We call such a list an *influenced dimension scenario*. As some scenarios may not be feasible, we may consider several of them. A set of scenarios will be translated to an influence constraint tree to be processed by the scheduler.

A dimension suitable for load/store vectorization with explicit vector types must meet several conditions: (a) it should be the innermost dimension of the loop nest; (b) its size must be 2 or 4 (3 is not supported yet and is left for future work) and (c) some (as many as possible) memory accesses are aligned and either constant or contiguous. Such conditions are softer than those to vectorize computation as we only target load/store operations: we may mix vector types with scalar types.

Algorihtm 2 depicts how to search for the set of influenced dimension scenarios I using the set of dimensions to schedule D for every statement in S. It starts with an empty set I. Influenced dimension scenarios I_s are progressively built from innermost to outermost (Lines 7 and 11) for each statement $s \in S$. At Line 8, $best()$ returns the dimension $d, d \in D \wedge d \notin I_s$ with the highest score according to the $cost()$ function which is detailed below. The selected dimension is added as the head of the list of dimensions I_s and the algorithm proceeds with the next outer dimension if necessary. Once enough scheduling dimensions have been added, I_s is added to the set I. The set I will then be used to build the constraint tree.

Algorithm 2: Build Influenced Dimension Scenarios

```
   Data: set of statements S, set of dimensions to schedule D,
         weight vector W, thread limit L
   Result: set of influenced dimension scenarios I
1  I := ∅ ;
2  forall s in S do
3      A := accesses of statement s ;
4      I_s := [ ] ;
5      depth := |D| ;
6      L_s := L ;
7      while |I_s| < 3 ∧ |I_s| < |D| do
8          b := best(W, D, A, L, I_s, depth) ;
9          I_s := [b, I_s] ;
10         L := L/size(b) ;
11         depth := depth − 1 ;
12     end
13     I := I ∪ I_s
14 end
```

The $cost()$ function to be used in $best()$ is:

$$cost(W, D, A, L, d) = w_1|V_w| + w_2|V_r| + \frac{w_3}{M} + w_4|C| + \frac{w_5 F L}{N}$$

where:

- $w_1, ..., w_n$ are coefficients from the weight vector W,
- N is the number of iterations for dimension d,
- $V_w \subset A$ (respectively $V_r \subset A$) is $\emptyset$ if $d < |D|$ or the subset of vectorizable store accesses (respectively load accesses) otherwise (favors store or load vectorization),
- M is the minimum stride on all accesses in A by dimension d (favors short memory jumps),
- $C \subset A$ is the subset of accesses with the minimum stride M at dimension d (favors as many references as possible with short memory jumps),
- F is 1 if $N < L$, 0 otherwise (last cost term favors high contribution to the number of threads not exceeding L).

The V_r and V_w subsets are used to favor load/store vectorization at the innermost dimension. In cases where only partial load/store vectorization is amenable, we favor coalescing by

minimizing the strides of accesses on remaining dimensions to schedule, while controlling the number of threads.

Through empirical studies, we observed better results when prioritizing vector types on write accesses over vector types on read accesses. The best weights configuration in our context is achieved with $w_1 = 5$, $w_2 = 3$ and other weight set to 1. $w1$ and $w2$ have higher values because load/store vectorization is the priority. To avoid big jumps in memory, $w3 = 1$ is sufficient as $w3/M$ quickly decreases as M grows. If multiple candidate dimensions enable strides of size 1, we favor the one with more accesses with stride 1. $w4 = 1$ is enough to order these dimensions. The same applies to $w5 = 1$ to order dimensions depending on thread use.

The next step is to build the constraints tree from the set of scenarios. Translating a given scenario to affine constraints for integration to the influence constraint tree is a straightforward process. Innermost scheduling coefficients corresponding to loop iterators should be equal to their coefficients in the last access function, while other coefficients are free (in our context, access functions are extremely simple, being either a constant or an affine form where only one loop iterator is present with coefficient 1, which further simplifies the process). Following scheduling dimensions should make sure scheduling coefficients corresponding to the same previously fixed coefficients are zero, while other coefficients are free, etc. Exploring all possible scenarios may lead to a combinatorial explosion. Our strategy is to select only few of the most profitable solutions (set to 8 in our experiments). Then for each scenario, we build higher priority variants influencing towards loop fusion (injecting constraints equating scheduling coefficients on first dimensions over multiple statements) and lower priority variants integrating less constraints (leaving free some scheduling functions for some statements). Finally the tree is built by considering common constraints to different scenarios, and using the cost function to order siblings.

In addition to constraint construction and constraint injection, our modified AKG version required two modifications.First, the mapping pass which decides about dimensions to be mapped to CUDA blocks and threads has been modified to avoid considering dimensions marked as vectorized. Second, a backend vectorization pass was added to actually translate the load/store vector loop to a version using explicit vector types.

VI. Experimental Evaluation

Our approach was implemented inside the tensor compiler AKG [6] and a modified version of the isl scheduler [32]. Experiments were conducted using MindSpore [7] r1.1, We compare our method to AKG using unmodified isl-0.22 scheduling, and the manual scheduling approach of TVM [33]. Tile sizes are selected by respective tool auto-tuners.

The experimental platform consists of an Intel Xeon CPU E5-2680 v4 (14 cores, 28 threads, clocked @ 2.40GHz, 756GB RAM), and a Nvidia Tesla V100 for PCIe (clocked @ 1245MHz, 16GB RAM), running Linux 4.15.0 on Ubuntu 18.04.5 LTS with CUDA 10.1 drivers. Linux FIFO scheduling is enabled and priority is set to 85. We profiled fused operators using `nvprof`. Experiments were executed 10 times and we use the corresponding mean values. Standard deviation for all reported values is below 5% of the corresponding mean values.

Table I lists the target end-to-end workloads we used to evaluate our method. The target end-to-end workload are from the ModelZoo of MindSpore. We limited ourselves to 5 epochs to avoid long run times.

TABLE I
Target End-to-End Workloads

Network	Type	Dataset
BERT [34]	nlp	zhwiki
LSTM [35]	nlp	ACLIMDB [35], GloVe [36]
MobileNetv2 [37]	cv	ImageNet [38]
ResNet-50 [39]	cv	CIFAR-10 [40]
ResNet-101 [39]	cv	ImageNet [38]
ResNeXt50 [41]	cv	ImageNet [38]
VGG16 [42]	cv	CIFAR-10 [40]

We compared four versions of the target end-to-end workloads: fused operators scheduled with standard isl scheduling (**isl**), fused operators optimized with TVM (**tvm**), influenced fused operators without enabling explicit load/store vectorization (**novec**) and influenced fused operators with vectorization (**infl**). We focus on total execution time of fused operators.

Table II reports fused operators statistics. Fused operator counts are shown in the corresponding columns: *total* is the total number of fused operators. *vec* refers to influenced fused operators eligible for load-store vectorization and *infl* is the total number of influenced fused operators. The seven next columns correspond to statistics when considering all operators while the remaining columns focus only on fused operators where our influence approach actually modified the scheduling with respect to isl's solution. Speedups are execution time speedups over fused operators scheduled with standard isl scheduling.

Our method does not impact all fused operators: it may happen that even without additional constraints, isl's solution is the same as ours, by chance or because of a limited solution space. From about half (on BERT) to about 90% (NASNet) of fused operators are influenced. Most of the influenced operators are eligible for explicit load/store vectorization. Our results show explicit load/store vectorization enabled by the influenced scheduling is consistently beneficial.

We can observe that our approach outperforms the reference on all networks with speedups up to 7.70x on overall fused operators or even 12.53x if only influenced operators are considered. Detailed analysis of fused operators show that networks with the highest speedups (e.g., resnet50, resnet101) are those involving many transpose operations. Those cases are challenging for the original isl scheduling that lacks cost models to take good scheduling decisions. They also correspond to cases where memory coalescing and explicit use of vector types mix well together. All in all, our approach shows positive effects and enables from modest to significant speedups on fused operators, contributing to exceed the performance of TVM's manual hand-tuned scheduling approach.

TABLE II
FUSED OPERATORS EXECUTION TIMES

				All Fused Operators							Influenced Fused Operators						
	Operator Count			Execution Time (ms)				Speedup			Execution Time (ms)				Speedup		
Network	total	vec	infl	isl	tvm	novec	infl	tvm	novec	infl	isl	tvm	novec	infl	tvm	novec	infl
BERT	109	53	53	33.63	183.83	35.58	32.05	0.18	0.95	1.05	12.41	12.31	14.37	10.81	1.01	0.86	1.15
LSTM	4	3	3	0.11	0.12	0.11	0.11	0.94	1.00	1.05	0.11	0.12	0.11	0.11	0.94	1.00	1.05
MobileNetv2	18	16	16	2.54	2.57	2.57	2.50	0.99	0.99	1.02	2.40	2.43	2.43	2.35	0.99	0.99	1.02
ResNet50	17	10	12	1.68	0.55	0.55	0.49	3.07	3.05	3.43	1.43	0.28	0.30	0.24	5.14	4.72	5.93
ResNet101	22	14	16	5.67	0.82	0.84	0.74	6.94	6.75	7.70	5.36	0.47	0.53	0.43	11.31	10.07	12.53
ResNeXt50	33	21	22	1.78	1.57	1.44	1.30	1.13	1.23	1.36	1.31	1.10	0.98	0.84	1.19	1.35	1.56
VGG16	14	9	10	3.22	2.96	2.55	2.27	1.09	1.26	1.42	3.08	2.81	2.41	2.12	1.09	1.28	1.45

VII. RELATED WORK

Automatic parallelization in the polyhedral model has been introduced by the seminal work of Feautrier on solving the affine scheduling problem to extract fine-grain parallelism [20], [21]. His integration of data dependencies and use of Farkas' lemma to linearize the problem remain strong foundations to subsequent work in the field. Lim and Lam built on his solution to extract coarse grain parallelism with outermost permutable loops [22]. Bondhugula et al. introduced the Pluto algorithm and its subsequent extension Pluto+, including an efficient linear cost function and an iterative construction to produce parallel, tilable codes that optimize data locality [23], [24]. Pouchet et al. proposed a different construction approach building on Vasilache's convex modeling of the semantically correct solution space [43], [44], allowing to build multidimensional scheduling in one step rather than using an iterative approach. Verdoolaege et al. proposed to mix Feautrier's approach with Pluto by using the former as a fallback to the latter when it fails at achieving perfect reuse distance optimization [12]. A number of efforts proposed to improve polyhedral scheduling by considering different cost models. Kong and Pouchet proposed a comprehensive set of linear cost functions to address various goals [45]. Several works addresses data access patterns, Vasilache et al.'s contiguity constraints for vectorization [46], Kong et al. to maximize stride-0/1 references [47], Zinenko et al. to exploit spatial locality [25], or Verdoolaege and Isoard consecutivity constraint [48]. With the exception to Zinenko et al.'s work which allows a non-linear decision on the cost function ordering [25], previous works make the best efforts and trade-offs to translate the objective function to affine constraints. Differently, we acknowledge non-linear decisions may provide useful guidance and means to dissociate non-affine decisions from affine result of the decision communicated though influence constraint trees.

Not integrated to the affine scheduling problem, other solutions addressing vectorization suggest, e.g., to drive directive-based loop transformations with a cost model [49] or to apply post-scheduling optimization [50]. Differently, our solution incorporates non-linear optimization criteria in a structured manner within polyhedral scheduling's global approach.

A number of frameworks have been developed to generate optimized codes for tensor computation on GPUs. Solutions such as TVM [33] or Tiramisu [51] decouple tensor computation from its scheduling while others integrate automatic scheduling approaches. Polyhedral scheduling integrated within the isl library [13], [32] and developed for the PPCG CUDA compiler [12] rely on a modified version of the Pluto algorithm and has been the foundation for several tensor compilers. Tensor Comprehensions [2] was arguably the first of them, building on Halide [52] and a modified version of PPCG to generate CUDA codes for tensor graphs. Both Diesel [3] and AKG [6] also build on isl scheduling used but provide specialized subsequent optimization passes to generate efficient codes. R-Stream [53] embeds a modified version of the Pluto algorithm with modification to benefit from tensor computation application domain properties. All those works may directly benefit from our approach and integrate additional objectives driving the scheduling construction rather than, e.g., developing specialized passes to solve issues. PolyAST+GPU [54] has its own particular polyhedral scheduling strategy to generate two levels of parallelism to address GPU architecture and uses a non-linear cost-model to reorder loops for memory coalescing. We believe the PolyAST+GPU strategy could be reproduced with constraint injection mechanism with the benefit of using a more versatile scheduling approach. Finally, cited related work address load/store vectorization using memory coalescing only while our approach also profitably exploit explicit vector types.

VIII. CONCLUSIONS AND FUTURE WORK

We presented an approach to better schedule instruction executions of AI/DL operators on GPU. To increase control over automatic polyhedral scheduling, we introduced the influence constraint tree abstraction that aims at providing multiple optimization scenarios decided by a non-linear optimizer freed of polyhedral model limitations. We modified a polyhedral scheduling algorithm to generate semantically correct solutions with the most profitable scenario. We built on this mechanism to improve scheduling quality in a production AI/DL framework when targeting GPU, by introducing load/store vectorization constraints exploiting both memory coalescing and vector types. Experimental results demonstrate the effectiveness of the approach, enabling performance speedups from few percents to significant improvements on all fused operators of typical DNNs. Ongoing work aims at leveraging constraint injection to optimize codes for NPU accelerators and to exploit cost function injection to further improve scheduling quality.

REFERENCES

[1] P. Feautrier and C. Lengauer, "Polyhedron model," in *Encyclopedia of Parallel Computing*. Springer US, 2011, pp. 1581–1592. ISBN 978-0-387-09766-4

[2] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. Devito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "The next 700 accelerated layers: From mathematical expressions of network computation graphs to accelerated gpu kernels, automatically," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 4, Oct. 2019. doi: 10.1145/3355606

[3] V. Elango, N. Rubin, M. Ravishankar, H. Sandanagobalane, and V. Grover, "Diesel: Dsl for linear algebra and neural net computations on gpus," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, ser. MAPL 2018, 2018, p. 42–51. doi: 10.1145/3211346.3211354

[4] T. Zerrell and J. Bruestle, "Stripe: Tensor compilation via the nested polyhedral model," *CoRR*, vol. abs/1903.06498, 2019.

[5] S. Tavarageri, A. Heinecke, S. Avancha, B. Kaul, G. Goyal, and R. Upadrasta, "PolyDL: Polyhedral optimizations for creation of high-performance DL primitives," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 1, Jan. 2021. doi: 10.1145/3433103

[6] 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*, Jun. 2021, p. 1233–1248. doi: 10.1145/3453483.3454106

[7] Huawei, "Mindspore," https://www.mindspore.cn/en, 2021.

[8] L. Chen and Y. Zeng, *Deep Learning and Practice with MindSpore*, ser. Cognitive Intelligence and Robotics. Springer Singapore, 2021. doi: 10.1007/978-981-16-2233-5

[9] J. Filipovič, M. Madzin, J. Fousek, and L. Matyska, "Optimizing cuda code by kernel fusion: Application on blas," *J. Supercomput.*, vol. 71, no. 10, p. 3934–3957, Oct. 2015. doi: 10.1007/s11227-015-1483-z

[10] M. Boehm, B. Reinwald, D. Hutchison, P. Sen, A. V. Evfimievski, and N. Pansare, "On optimizing operator fusion plans for large-scale machine learning in SystemML," *Proc. VLDB Endow.*, vol. 11, no. 12, p. 1755–1768, Aug. 2018. doi: 10.14778/3229863.3229865

[11] B. Qiao, O. Reiche, F. Hannig, and J. Teich, "From loop fusion to kernel fusion: A domain-specific approach to locality optimization," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Washington, USA, Feb. 2019, pp. 242–253. doi: 10.1109/CGO.2019.8661176

[12] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for CUDA," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 54:1–54:23, 2013. doi: 10.1145/2400682.2400713

[13] S. Verdoolaege and G. Janssens, "Scheduling for ppcg," *CW Reports*, 06 2017. doi: 10.13140/RG.2.2.28998.68169

[14] W. Kelly and W. Pugh, "A unifying framework for iteration reordering transformations," in *IEEE Intl. Conf. on Algorithms and Architectures for Parallel Processing (ICAPP'95)*, Apr. 1995, pp. 153–162. doi: 10.1109/ICAPP.1995.472180

[15] C. Chen, J. Chame, and M. Hall, "CHiLL: A framework for composing high-level loop transformations," USC Computer Science, Tech. Rep. 08-897, June 2008.

[16] L. Bagnères, O. Zinenko, S. Huot, and C. Bastoul, "Opening polyhedral compiler's black box," in *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Barcelona, 2016. doi: 10.1145/2854038.2854048

[17] F. Quilleré, S. Rajopadhye, and D. Wilde, "Generation of efficient nested loops from polyhedra," *International Journal of Parallel Programming*, vol. 28, no. 5, pp. 469–498, Oct. 2000. doi: doi.org/10.1023/A:1007554627716

[18] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, Juan-les-Pins, France, Sep. 2004, pp. 7–16. doi: 10.5555/1025127.1025992

[19] T. Grosser, S. Verdoolaege, and A. Cohen, "Polyhedral AST generation is more than scanning polyhedra," *ACM Trans. Program. Lang. Syst.*, vol. 37, no. 4, pp. 12:1–12:50, 2015. doi: 10.1145/2743016

[20] P. Feautrier, "Some efficient solutions to the affine scheduling problem: one dimensional time," *International Journal of Parallel Programming*, vol. 21, no. 5, pp. 313–348, october 1992. doi: 10.1007/BF01407835

[21] ——, "Some efficient solutions to the affine scheduling problem, part II: multidimensional time," *International Journal of Parallel Programming*, vol. 21, no. 6, pp. 389–420, Dec. 1992. doi: 10.1007/BF01379404

[22] A. Lim, "Improving parallelism and data locality with affine partitioning," Ph.D. dissertation, Stanford University, 2001. doi: 10.5555/934599

[23] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proc. of the 2008 ACM Conf. on Programming language design and implementation (PLDI'08)*, Tucson, AZ, USA, Jun. 2008. doi: 10.1145/1379022.1375595

[24] U. Bondhugula, A. Acharya, and A. Cohen, "The pluto+ algorithm: A practical approach for parallelization and locality optimization of affine loop nests," *ACM Trans. Program. Lang. Syst.*, vol. 38, no. 3, pp. 12:1–12:32, 2016. doi: 10.1145/2896389

[25] O. Zinenko, S. Verdoolaege, C. Reddy, J. Shirako, T. Grosser, V. Sarkar, and A. Cohen, "Modeling the conflicting demands of parallelism and temporal/spatial locality in affine scheduling," in *Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria*, 2018, pp. 3–13. doi: 10.1145/3178372.3179507

[26] F. Irigoin and R. Triolet, "Computing dependence direction vectors and dependence cones with linear systems," Ecole des Mines de Paris, Fontainebleau (France), Tech. Rep. ENSMP-CAI-87-E94, 1987.

[27] U. Bondhugula, "Effective Automatic Parallelization and Locality Optimization using the Polyhedral Model," Ph.D. dissertation, The Ohio State University, August 2008. doi: 10.5555/1559029

[28] A. W. Lim and M. S. Lam, "Maximizing parallelism and minimizing synchronization with affine transforms," in *24th ACM Symposium on Principles of Programming Languages*, Paris, France, Jan. 1997, pp. 201–214. doi: 10.1145/263699.263719

[29] M. T. Kandemir, I. Kadayif, A. N. Choudhary, and J. Zambreno, "Optimizing inter-nest data locality," in *CASES Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, October 2002, pp. 127–135. doi: 10.1145/581630.581650

[30] W. Li and K. Pingali, "A singular loop transformation framework based on non-singular matrices," *International Journal of Parallel Programming*, vol. 22, no. 2, pp. 183–205, April 1994. doi: 10.1007/BF02577874

[31] A. Schrijver, *Theory of linear and integer programming*. John Wiley & Sons, 1986. doi: 10.5555/17634

[32] S. Verdoolaege, "*isl*: An integer set library for the polyhedral model," in *ICMS 2010, Third International Congress on Mathematical Software*, Kobe, Japan, Sep. 2010, pp. 299–302. doi: 10.1007/978-3-642-15582-6_49

[33] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: An automated end-to-end optimizing compiler for deep learning," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18, 2018, p. 579–594. doi: 10.5555/3291168.3291211

[34] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. doi: 10.18653/v1/N19-1423

[35] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning word vectors for sentiment analysis," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. Portland, Oregon, USA: Association for Computational Linguistics, Jun. 2011, pp. 142–150. doi: 10.5555/2002472.2002491

[36] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543. doi: 10.3115/v1/D14-1162

[37] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520. doi: 10.1109/CVPR.2018.00474

[38] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255. doi: 10.1109/CVPR.2009.5206848

[39] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778. doi: 10.1109/CVPR.2016.90

[40] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," University of Toronto, Tech. Rep., 2009.

[41] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," 2017.

[42] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2015.

[43] N. Vasilache, "Scalable optimization techniques in the polyhedral model," PhD Thesis, University Paris-Sud 11, Orsay, France, September 2007.

[44] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache, "Loop transformations: Convexity, pruning and optimization," in *38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (PoPL'11)*, Austin, TX, Jan. 2011, pp. 549–562. doi: 10.1145/1925844.1926449

[45] M. Kong and L. Pouchet, "Model-driven transformations for multi- and many-core cpus," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, K. S. McKinley and K. Fisher, Eds., 2019, pp. 469–484. doi: 10.1145/3314221.3314653

[46] N. Vasilache, B. Meister, M. Baskaran, and R. Lethin, "Joint scheduling and layout optimization to enable multi-level vectorization," in *In Second Int. Workshop on Polyhedral Compilation Techniques (IMPACT 2012)*, 2012.

[47] M. Kong, R. Veras, K. Stock, F. Franchetti, L. Pouchet, and P. Sadayappan, "When polyhedral transformations meet SIMD code generation," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, H. Boehm and C. Flanagan, Eds., 2013, pp. 127–138. doi: 10.1145/2491956.2462187

[48] S. Verdoolaege and A. Isoard, "Extending pluto-style polyhedral scheduling with consecutivity," in *In 8th Int. Workshop on Polyhedral Compilation Techniques (IMPACT 2018)*, 2018.

[49] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen, "Polyhedral-model guided loop-nest auto-vectorization," in *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '09, 2009, p. 327–337. doi: 10.1109/PACT.2009.18

[50] S. Moll, J. Doerfert, and S. Hack, "Input space splitting for opencl," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC 2016, 2016, p. 251–260. doi: 10.1145/2892208.2892217

[51] R. Baghdadi, J. Ray, M. B. Romdhane, E. D. Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. P. Amarasinghe, "Tiramisu: A polyhedral compiler for expressing fast and portable code," in *IEEE/ACM International Symposium on Code Generation and Optimization, CGO*, Feb. 2019, pp. 193–205. doi: 10.1109/CGO.2019.8661197

[52] 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 *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13, 2013, p. 519–530. doi: 10.1145/2491956.2462176

[53] B. Meister, E. Papenhausen, and B. Pradelle, "Polyhedral tensor schedulers," in *17th International Conference on High Performance Computing & Simulation, HPCS*, Jul. 2019, pp. 504–512. doi: 10.1109/HPCS48598.2019.9188233

[54] J. Shirako, A. Hayashi, and V. Sarkar, "Optimized two-level parallelization for gpu accelerators using the polyhedral model," in *Proceedings of the 26th International Conference on Compiler Construction*, ser. CC 2017, Feb. 2017, p. 22–33. doi: 10.1145/3033019.3033022