SIRIUS: HARVESTING WHOLE-PROGRAM OPTIMIZATION OPPORTUNITIES FOR DNNS

Yijin Li [*1 2] Jiacheng Zhao [*1 2 3] Qianqi Sun [1 2] Haohui Mai [4] Lei Chen [1 2] Wanlu Cao [1 2] Yanfan Chen [1 2]
Zhicheng Li [1 2] Ying Liu [1 3] Xinyuan Zhang [1] Xiyu Shi [1] Jie Zhao [5] Jingling Xue [6] Huimin Cui [1 2]
Xiaobing Feng [1 2 3]

ABSTRACT

As emerging applications are rapidly moving to accelerators, a great deal of research has been proposed to improve the performance of the accelerators. For the AI applications, fruitful software-driven research has been focused on proposing new programming languages, new kernel fusion heuristics, new optimization tuning approaches, and new software execution engines. However, how to leverage classical compiler optimizations to generate efficient code is an overlooked aspect of performance. In this paper, we propose a whole-program analysis and optimization compiler framework, SIRIUS, to uniformly model the host and kernel computations in a unified polyhedral representation and, further, seek maximal fusion opportunities from the global view so that the fused kernel can benefit from classical optimizations. Evaluations over representative DNN models demonstrate that SIRIUS can achieve up to $11.98\times$ speedup over TensorRT, and $154.84\times$ speedup over TensorFlow. In particular, for BERT, SIRIUS can achieve $1.46\times$ speedup over TensorRT.

1 INTRODUCTION

Emerging applications such as intelligent virtual assistants (Brown et al., 2020), autonomous driving (Zoph et al., 2018), and computer-aided drug designs (Jia et al., 2020) have been transforming the ways we live, work and thrive. All these applications are powered by the massive, readily available computation powers from dedicated accelerators such as GPUs, TPUs, and neural engines (Jia et al., 2019b; Jouppi et al., 2017; Liao et al., 2019; Rocki et al., 2020; Zhao et al., 2019a). Therefore, their performance is essential to unlocking innovations that reshape our society in the next decade.

The fast evolution of accelerator applications (Amodei et al., 2016; Fu et al., 2019; Gupta et al., 2020; Hemmat et al., 2020; Hochreiter & Schmidhuber, 1997; Judd et al., 2016; Krizhevsky et al., 2017; Kwon et al., 2019; Ning et al., 2019; Qin et al., 2020; Sutskever et al., 2014; Xie et al., 2017; Zoph et al., 2018) and hardware architectures (González-Alvarez et al., 2016; Jia et al., 2019b; Jouppi et al., 2017; Liao et al., 2019; NVIDIA Corporation., 2017; Rocki et al., 2020; Zhao et al., 2019a) present unique challenges to optimizing compilers. Accelerator applications, particularly deep neural network (DNN) applications are semi-structured but also convoluted. An optimizing compiler must generate performant implementations of primitive operators, untangle the convoluted dependency of the network and properly stage the network down to a streamlined, tailored implementation for specific hardware to deliver the required performance.

Numerous research has been conducted on improving the performance of DNN networks via extremely manually optimizations (Ahmed et al., 2022), developing high-performance libraries of primitive operators (NVIDIA Corporation., a;b), providing supports from programming languages (Abadi et al., 2016; Baghdadi et al., 2019; Chen et al., 2018a; Hagedorn et al., 2020; Liu et al., 2022; Ma et al., 2020; Ragan-Kelley et al., 2013), fusing GPU kernels of different layers (Ma et al., 2020; Wahib & Maruyama, 2014; Wang et al., 2021; Zhao et al., 2022; Zheng et al., 2020b;c; 2022), and choosing the performant candidates via auto tuning (Chen et al., 2018a; Jia et al., 2019a; Zhang et al., 2020; Zheng et al., 2020b). Experience shows that kernel fusions are able to reduce the synchronization overheads between the host and the accelerator, extract parallelism via aligning the data dependency of the network with the hardware execution units (Ma et al., 2020; Niu et al., 2021), and utilize memory bandwidths via pipelining the computations (Zhao et al., 2021; 2022; Zheng et al., 2022).

Rather than seeking fusion opportunities starting from the graph and operator representations in existing AI ecosystems (including programming framework, e.g., TensorFlow/PyTorch, and AI compilers, e.g., TVM), we investigate

[*]Equal contribution [1]SKLP, Institute of Computing Technology, CAS [2]University of Chinese Academy of Sciences [3]Zhongguancun Laboratory [4]Hengmuxing Technologies [5]State Key Laboratory of Mathematical Engineering and Advanced Computing [6]University of New South Wales. Correspondence to: Ying Liu <liuying2007@ict.ac.cn>.

Proceedings of the 6th MLSys Conference, Miami, FL, USA, 2023. Copyright 2023 by the author(s).

the performance potential from the perspective of classical compiler optimizations, by starting from all the source codes of DNN models, enhancing the inter-procedural and whole-program analysis towards heterogeneous programming models, seeking fusion opportunities from the global view and leveraging existing compiler optimization techniques to improve the performance of DNN models.

Figure 1(a) shows a sub-graph selected from `BERT` (Devlin et al., 2018) and (b) shows its performance characteristics. The sub-graph includes three branches sharing one common `input` matrix, corresponding to the `QKV` computations in the self-attention mechanism. For each branch, the `GEMM` operator is followed by an `add` operator and a `reshape` operator. We discuss the implementations of the sub-graph in state-of-the-art deep learning systems as follows:

- *No-fusion (TensorFlow).* The circled sub-graph has 9 operators in total, i.e., 9 kernels with $4,715,712$ instructions, and its execution time is 54.00μs.
- *Apollo fusion.* For this sub-graph, Apollo (Zhao et al., 2022) would fuse the `GEMM` and `add` together and reduce the number of operators to 6 with $3,818,016$ instructions, by leveraging partition-based fusion, and its execution time is 46.20μs.
- *TensorRT fusion.* TensorRT (TRT) (NVIDIA Corporation., a) manually fused the three `GEMM` and `add` operators together, reduces the number of operators to 4 with $3,337,632$ instructions, thus the execution time is 29.91μs.
- SIRIUS *(our system).* After our whole-program analysis, we can fuse *all* the 9 kernels into one and reduce the number of instructions to $769,536$, and reduce the execution time to 15.82μs.

Furthermore, we can perform a number of classical whole-program compiler optimizations, and we discuss two optimizations here. First, we can identify that there exist three load operations for one common read-only data `input` thus two of them can be eliminated by leveraging the redundant load elimination algorithm. Second, we can perform SSA-based dead code elimination on the fused instructions from `GEMM`, `add`, and `reshape`. In particular, as shown by the pseudo-code in Figure 1(c), three branches with 27 pseudo instructions(each of them executed 9 pseudo instructions), would be optimized into 13 pseudo instructions.

To explore to which extent accelerator applications can benefit from whole-program analysis and optimizations for host and kernels, we have designed and implemented SIRIUS, a compiler framework that coordinately optimizes parallel (by exploiting parallelism between threads) and sequential performance (by increasing the performance of each thread) for accelerator applications. SIRIUS takes the CUDA kernels and the driver codes from the host side as inputs, explores different fusion schedules to optimize for both parallel and sequential performance of the applications, and finally outputs the optimized, transformed program. SIRIUS exploits three simple heuristics: (1) propagating information across the host and device kernels, (2) greedily maximizing the fusion schedule that would not introduce global synchronizations, and (3) preferring a fusion schedule that merges the same fragments in close proximity to facilitate the thread organization.

We have evaluated SIRIUS using 6 representative deep neural networks on an NVIDIA A100 system and an NVIDIA V100 system. On the A100 system, SIRIUS achieves $1.46\times$ speedup over TensorRT for BERT.

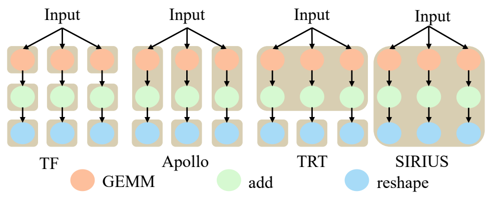
This paper makes the following contributions:

1. It describes the performance profiles for representative accelerator applications. It identifies that enlarging the scope for code optimization can have significant impacts on the performance of the accelerator kernels.
2. It unifies the data dependencies and synchronizations from both the host and device sides of the applications on a unified polyhedral abstract representation, which enables performing inter-procedural optimizations across the host driver and multiple device kernels.
3. It proposes two heuristics to globally seek greedy fusion opportunities and arrange the code fragments after fusion. It demonstrates their effectiveness quantitatively on 6 representative deep neural networks.
4. It achieves an average of $4.32\times$ (up to $7.12\times$) speedup over TensorRT for the 6 DNN models. In particular, for BERT, SIRIUS achieves $1.46\times$ speedup over TensorRT.

2 OVERVIEW

Figure 2 describes the overall architecture of SIRIUS. SIRIUS first takes CUDA programs as inputs and compiles both the host and device sides of a program into LLVM IR. Second, SIRIUS leverages whole-program analysis to capture the data and control dependency for the LLVM IRs of the whole program on the unified abstract representations (UAR) (Section 3). UAR represents the dependency in the polyhedral domains (Bondhugula et al., 2008; Zhao & Di, 2020) to describe the fine-grain memory layouts and access patterns.

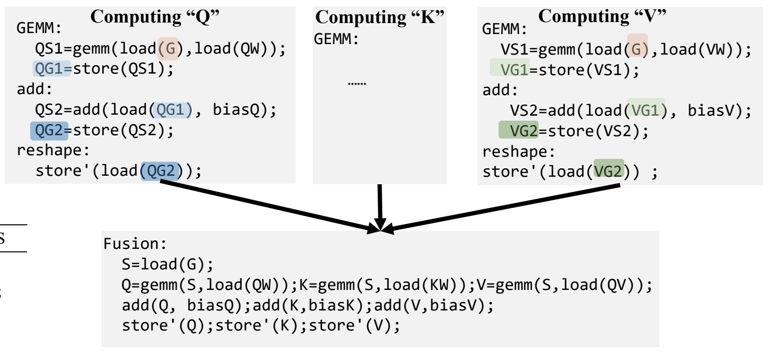
To precisely represent the data dependency across kernels, UAR annotates such a data dependency with its *width* to indicate the scope that carries the dependency. SIRIUS defines three types of level annotations, i.e., thread-level, block-level, and global. In particular, a *thread/block-level data dependency* means the dependency is one-to-one from the thread/block of the source kernel to the thread/block of the



(a) Fusion results by TF/Apollo/TRT/SIRIUS(our approach)

	TF	Apollo	TRT	SIRIUS
#of Kernels	9	6	4	1
#of Threads	1,022,976	912,384	898,560	12,288
Runtime(μs)	54.00	46.20	29.91	15.82
Total SASS insts.(10^5)	47.15	38.18	33.38	7.69

(b) Performance characteristics



Computing "Q"

```
GEMM:
  QS1=gemm(load(G),load(QW));
  QG1=store(QS1);
add:
  QS2=add(load(QG1), biasQ);
  QG2=store(QS2);
reshape:
  store'(load(QG2));
```

Computing "K"

```
GEMM:

    ......
```

Computing "V"

```
GEMM:
  VS1=gemm(load(G),load(VW));
  VG1=store(VS1);
add:
  VS2=add(load(VG1), biasV);
  VG2=store(VS2);
reshape:
store'(load(VG2)) ;
```

```
Fusion:
  S=load(G);
  Q=gemm(S,load(QW));K=gemm(S,load(KW));V=gemm(S,load(QV));
  add(Q, biasQ);add(K,biasK);add(V,biasV);
  store'(Q);store'(K);store'(V);
```

(c) SIRIUS: pseudo-code before and after optimizations

Figure 1. Motivation example (a sub-graph from `BERT`).

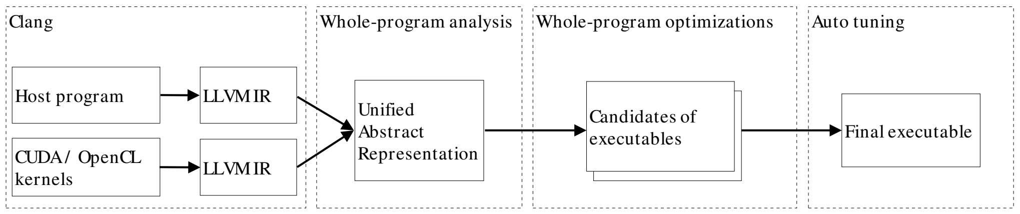


Figure 2. Overall architecture of SIRIUS.

target kernel. A thread-level data dependency means the two kernels can be fused without introducing any synchronization statements, while a block-level data dependency means a `__syncthreads` is required when fusing these two kernels. A global dependency means that there exist cross-block dependencies between two kernels, e.g. the input data of a subsequent kernel's block is the output data of multiple blocks in the prior kernel. Figure 3(a) shows the pseudo-code for the sub-graph of BERT discussed in Figure 1, and Figure 3(b) shows the block-level, thread-level, and global dependencies.

The key insight for fusion in SIRIUS has three principles.

- **(Principle 1 (P1)).** If two kernels have only thread/block-level dependencies, they can be fused locally by merging the kernel statements in order.
- **(Principle 2 (P2)).** If two kernels have global dependencies, fusion is disabled due to the fact that global synchronizations are poorly supported by GPUs (Harris et al., 2007; Xiao & Feng, 2010).
- **(Principle 3 (P3)).** If two kernels have no dependencies, SIRIUS fuses them together only when they have high code similarity (we restrict they are the same in this paper).

SIRIUS computes a greedy fusion schedule using two passes, i.e., a local fusion followed by a global fusion. The local fusion pass is implemented as a BFS traverse on UAR, and the global fusion pass is implemented using the ISL solver by introducing a schedule constraint that specifies the execution order of two kernels with global data dependencies in UAR. For Figure 3(b), the `GEMM(2)` is not fused due to the global data dependency, and the three branches are fused since they have no dependencies and have the same code.

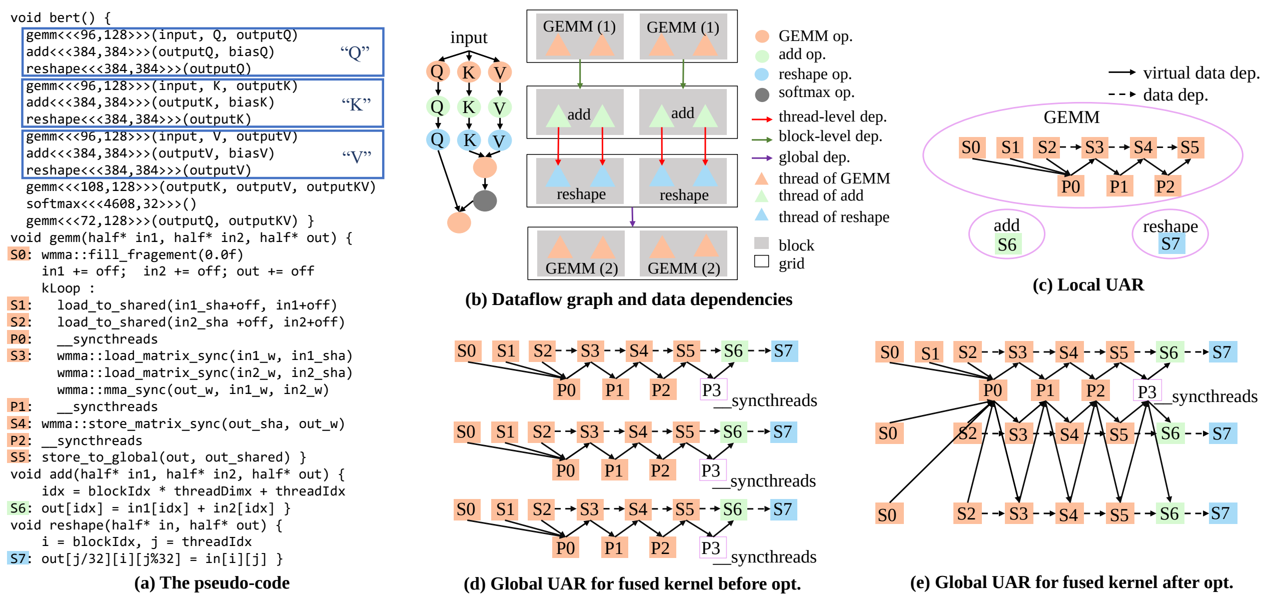
SIRIUS fuses the invocations of kernels from the output schedule. Kernels can be arranged horizontally (i.e., executing the fragments in different blocks for parallelism) or vertically (i.e., executing the fragments in the same thread for sequential performances). Essentially different arrangements present different tradeoffs between parallelism and sequential performance. The optimal tradeoff is highly dependent on hardware configurations. Therefore SIRIUS uses an auto tuner to choose the performant schedule and reports it to the users.

3 CONSTRUCTING THE UNIFIED ABSTRACTION REPRESENTATION (UAR)

UAR describes the synchronizations and data dependency of both the host driver and device kernels on a single, unified dependency graph. It represents the dependency as affine relations in polyhedral domains to capture the dependency and access patterns. The information of UAR drives interprocedural analysis and transformations in SIRIUS.

3.1 Backgrounds

This subsection provides basic backgrounds on polyhedral analysis (Baghdadi et al., 2019; Bondhugula et al., 2008;



```
void bert() {
  gemm<<<96,128>>>(input, Q, outputQ)
  add<<<384,384>>>(outputQ, biasQ)              "Q"
  reshape<<<384,384>>>(outputQ)
  gemm<<<96,128>>>(input, K, outputK)
  add<<<384,384>>>(outputK, biasK)              "K"
  reshape<<<384,384>>>(outputK)
  gemm<<<96,128>>>(input, V, outputV)
  add<<<384,384>>>(outputV, biasV)              "V"
  reshape<<<384,384>>>(outputV)
  gemm<<<108,128>>>(outputK, outputV, outputKV)
  softmax<<<4608,32>>>()
  gemm<<<72,128>>>(outputQ, outputKV) }
void gemm(half* in1, half* in2, half* out) {
S0: wmma::fill_fragement(0.0f)
    in1 += off;  in2 += off; out += off
    kLoop :
S1:   load_to_shared(in1_sha+off, in1+off)
S2:   load_to_shared(in2_sha +off, in2+off)
P0:   __syncthreads
S3:   wmma::load_matrix_sync(in1_w, in1_sha)
      wmma::load_matrix_sync(in2_w, in2_sha)
      wmma::mma_sync(out_w, in1_w, in2_w)
P1:   __syncthreads
S4: wmma::store_matrix_sync(out_sha, out_w)
P2: __syncthreads
S5: store_to_global(out, out_shared) }
void add(half* in1, half* in2, half* out) {
    idx = blockIdx * threadDimx + threadIdx
S6: out[idx] = in1[idx] + in2[idx] }
void reshape(half* in, half* out) {
    i = blockIdx, j = threadIdx
S7: out[j/32][i][j%32] = in[i][j] }
```

Figure 3. Workflow of SIRIUS for the sub-graph in BERT.

Bondhugula, 2008; Grosser et al., 2012) and the call graph.

A dependency graph $G = (V, E)$ is a directed multigraph that records the data and control dependencies for a particular procedure. A vertex $v \in V$ represents a statement in the procedure. An iteration vector $\vec{i} = (i_0, i_1, \ldots, i_n) \in \mathcal{D}^v$ describes the values of the indices of all loops surrounding the vertex v. An edge $e = (S_i, S_j) \in E$ defines an edge from statement S_i to statement S_j, which represents a polyhedral dependence from a dynamic instance of S_i to an instance of S_j. The polyhedral dependence is characterized by the *dependence polyhedron* $\mathcal{P}_e$ (Bondhugula et al., 2008; Bondhugula, 2008) that captures the exact dependence information of e based on iteration vectors. If $\vec{s}$ and $\vec{t}$ are the source and target iterations that are dependent, we can express

$$\langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_e \iff \vec{s} \in \mathcal{D}^{S_i}, \vec{t} \in \mathcal{D}^{S_j}$$
$$\text{are dependent through edge } e \in E$$

3.2 UAR

UAR is defined as a dependency graph G with additional information for modeling host driver and device kernels in a unified manner. In UAR, the vertex $v \in V$ is characterized by the memory effect of the host or kernel statement represented by v, and the edge e is characterized by the dependence polyhedron $\mathcal{P}_e$, which describes the synchronizations and data dependency between dynamic instances of statements of both the host driver and device kernels.

SIRIUS constructs the UAR in two phases. SIRIUS first performs a local analysis for each function. It analyzes the dependency and effects of each statement. Second, SIRIUS traverses the call graph from the bottom to propagate the results from local analysis to each call site. Ultimately, it results in a unified graph describing the program's dependency.

3.3 Local analysis

The local analysis phase performs two functionalities, i.e., performing dependency analysis for each function and computing memory access effect of each function.

Analyzing local dependency. SIRIUS augments the dependency analysis in LLVM (Lattner & Adve, 2004) in two ways to compute the local dependency graph for each function. First, SIRIUS introduces loops in device functions to represent the parallelism for both the blocks and the threads.[1] Second, if SIRIUS is able to show that quasi-affine accesses (such as modulo expressions) only happen inside the same block, SIRIUS rewrites the quasi-affine accesses into affine by replacing the modulo expression using its value range to compute the memory effects.

SIRIUS does not resolve the call sites during the phase of local analysis. It marks the call site with unknown effects and adds edges for control dependency to prevent reordering the call sites. SIRIUS treats synchronization barriers like `__syncthreads` as special function calls. It adds the edges for control dependency but annotates it with empty effects.

As no data dependencies exist between a computation statement and synchronization barriers, it is not straightforward to explicitly express their execution order in the polyhe-

[1] SIRIUS does not support `syncwarp()` yet as we found it is not widely used. Warps can be modeled in a similar way.

dral model. However, the synchronization barriers intimate implicit control dependencies to the program, we thus introduce a virtual data dependency between a synchronization barrier and the execution statements by converting the implicit control dependencies into data dependencies, and we denote such dependencies as *virtual data dependencies*.

SIRIUS follows a similar workflow for procedures on the host, except that it recognizes intrinsics such as memory operations, including copies and mallocs, as well as synchronization primitives (e.g., `cudaDevice-Synchronize()`) and annotates them accordingly.

Computing memory effect. For each kernel, SIRIUS extends the scalar evolution algorithm in LLVM to compute its memory effect at the level of thread, block and kernel. In particular, the memory effect for a kernel is represented as a read-set and a write-set representing the memory locations that are read and written by the kernel, at the level of thread-/block/kernel. SIRIUS invokes the *ScalarEvolutionAnalysis* pass to compute the thread-level memory effect directly. For each thread block, SIRIUS aggregates the memory effects of all threads in the block on the dimension of thread ID, and obtains the block-level memory effect. Similarly, SIRIUS aggregates the memory effects of all thread blocks on block ID dimension, obtaining the kernel-level memory effect.

3.4 Bottom-up analysis

The bottom-up analysis propagates the dependency and the effects along the call graph to compute the UAR for the whole program, and determines the polyhedra of each data dependency together with its dependency width.

SIRIUS propagates the information via cloning. Starting from the bottom of the call graph, the bottom-up analysis resolves the parameters for the call sites, inlines the local UAR into the UAR of the caller, and determines the data dependency width. When resolving the call sites of kernel launches, SIRIUS inlines only the callee's effects on the global memory heaps instead of its UAR, thus it discards the effects on shared memory which are irrelevant on the host side.

By examining the memory effect of each call site, SIRIUS enhances the dependency analysis algorithm in LLVM to compute the dependency polyhedra $\mathcal{P}_e$ and annotates it with the tag of *thread-level*, *block-level* or *global*. For a data dependency $e_{<s,t>}$, its width is determined by the level that carries the dependency. For example, consider the dependency caused by an array A which is written by s and read by t, for s we use $A_s^{thread(tid)}$, $A_s^{block(bid)}$ and A_s to represent the access region of s's one thread, one thread block and the whole kernel respectively, and similar for t. The dependency width of e is defined as block-level, only if the access regions from the blocks of s are linear disjoint

$$\forall(i,j) A_s^{block(i)} \cap A_s^{block(j)} = \emptyset$$

and the access region from each block of t depends on only one block of s

$$\forall j, \exists! i, A_s^{block(i)} \cap A_t^{block(j)} \neq \emptyset$$

The width of thread-level and global are defined similarly.

Currently, we have not implemented the support of indirect or recursive calls in SIRIUS. Our experience shows that the majority of accelerator programs rely on simple direct calls. Supporting recursive calls would allow SIRIUS to analyze use cases such as depth-first search which has irregular patterns. It is possible to implement the supports using approaches described in data-structure analysis (Lattner et al., 2007).

Example. Figure 3 (a) describes the implementation of the BERT sub-graph, (c) shows the local UAR of `GEMM`, `add` and `reshape`, and (d) shows the top-level UAR. Solid and dashed arrows represent the virtual and the real data dependency. SIRIUS also annotates the edges of virtual dependency with corresponding polyhedra and width annotation.

3.5 Alias Analysis

The key challenge in our alias analysis is to track and resolve the mappings between the host handles and the actual regions of the device memories. SIRIUS first categorizes the pointers of the whole program into host pointers and device pointers, analyzes the `cudaMemcpy` statements to ensure that there does not exist alias across host and device pointers, and then performs alias analysis for the host pointers and device memories separately.

To get more precise analysis results, SIRIUS extends the Andersen's algorithm (Andersen, 1994) and implements a field-/array-/context-sensitive path-insensitive inclusion-based points-to algorithm. SIRIUS leverages the approaches proposed by (Pearce et al., 2007; Whaley & Lam, 2004) for implementing the field-/array-/context-sensitivity in LLVM.

The analysis is accurate enough for DNN models due to their regular memory accesses. For DNN models, the memory effect analysis and dependence analysis may introduce some inaccuracy. For example, if a kernel contains a modulo operation to compute the array subscript, SIRIUS would change the modulo operation into affine by using its value range. Therefore, the memory effect analysis would report an amplified result. For our DNN benchmarks, the inaccuracy would not introduce further dependency analysis and optimizations.

4 DERIVING THE FUSION SCHEDULES

4.1 Computing a greedy fusion schedule

As discussed in Section 2, we introduced three principles for computing the fusion schedule and implementing them using a local fusion pass followed by a global fusion pass.

For Principle (P1), the local fusion pass performs a BFS traverses to the top-level UAR. For each node, if it carries only thread-level or block-level data dependency to its predecessor, SIRIUS would merge them into one node, and their execution order can be guaranteed by placing the statements in order and inserting `__syncthreads` for block-level dependencies.

For Principle (P2) and (3), SIRIUS leverages the ISL solver to compute a fusion schedule over the top-level UAR. SIRIUS generates the relations and feeds the relations to the ISL solver, then the solver would determine a schedule by modeling the validity relations into ILP constraints (Verdoolaege et al., 2017; Zinenko et al., 2017).

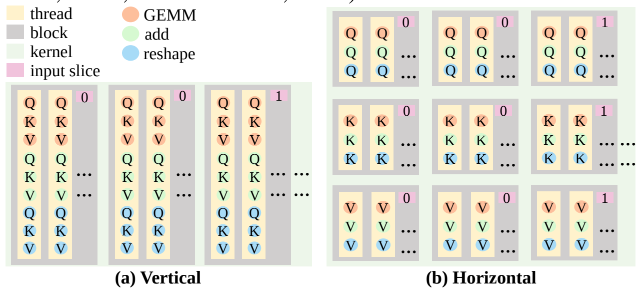


Figure 4. Vertical and horizontal kernel arrangement.

First, SIRIUS leverages the validity relations to represent Principle (P2) by introducing a schedule constraint to specify the execution order of two kernels with global data dependencies in UAR. In particular, only the global data dependencies would be modeled as constraints and fed to the ISL solver. If two kernels have thread-level/block-level data dependencies, they have been merged into one during the local fusion pass.

4.2 Arranging kernel fragments - horizontal and vertical

Given a set of kernel fragments without dependencies, SIRIUS can fuse them in two possible directions. As shown in Figure 4, it can fuse the kernel either horizontally (i.e., executing the fragments in different blocks) or vertically (i.e., executing the fragments in the same thread). The two directions represent tradeoffs between parallelism and opportunities for sequential optimizations. The horizontal fusion would increase the parallelism. However, the vertical fusion decreases the parallelism but it can open up more opportunities for post-fusion compiler optimizations. Therefore, SIRIUS generates a series versions of codes from fully horizontal to fully vertical (with some hybrid versions in-between), applies a number of optimizations described in Section 5, and leverages auto-tuning to select the performant version.

When generating code for vertical fused fragments, SIRIUS concatenates the shared memory buffers to utilize the shared memory more efficiently. When fusing fragments with different iteration domains (i.e., blocks and threads), SIRIUS aligns them either to the maximal or the minimal domain and generates corresponding `if` or `for` statements accordingly. When a fuse involves an element-wise kernel, SIRIUS adjusts its iteration domain according to other kernels thus the element-wise statements can be embedded into the iteration domain of the fused kernel. For horizontal fusion code generation, SIRIUS reconfigures the gridDim and BlockDim for the host and emits branches to different kernels for the device.

Example. SIRIUS detects that the nine operators as the fusion candidates in Figure 3, and the local fusion pass fuses the `GEMM`, `add` and `reshape` vertically and inserts `__syncthreads` between `GEMM` and `add` to ensure the execution order during the BFS traverse. The three nodes of UAR would be merged into one, denoted as LFK. Then the global fusion pass leverages the ISL solver and obtains the schedule that the three LFK nodes are fused since they have no dependencies and have the same code.

Furthermore, as shown in Figure 4, the ”Q/K/V“ three branches be fused from fully horizontal to fully vertical. In Figure 4(a), one input slice is executed in multiple blocks and each thread executes the ”GEMM+add+reshape” computing sequence for Q, K **and** V. In Figure 4(b), each block executes ”GEMM+add+reshape” computing sequence for Q **or** K **or** V. SIRIUS selects the fully vertically fused version as the performant one via auto-tuning. Section 7 discusses these tradeoff between parallelism and sequential optimizations in detail.

5 OPTIMIZING FUSED KERNELS

Algorithm 1 Optimizing synchronizations

```
Input: uar
Result: uar
1  for vdep in uar.vdeps do
2      for ddep in uar.ddeps do
3          if No_VDep_Path(ddep.src, ddep.dst) then
4              uar.remove(cdep);
5  for vdep in uar.vdeps do
6      for ddep in uar.ddeps do
7          if Exist_VDep_Path_BypassV(ddep.src, ddep.dst, vdep)
            then
8              uar.remove(cdep);
9  for pnode in uar.cdeps do
10     if pnode.out_degree == 0 then
11         uar.remove(pnode);
12 return uar
```

With the derived fusion schedule, SIRIUS merges the local UARs together and gets the fused UAR. For fragments that are horizontally fused, SIRIUS does not introduce extra synchronizations since the fragments are executed by different blocks. For fragments that are vertically fused, SIRIUS inserts a synchronization node between these fragments if there exist block-level data dependencies.

In the fused UAR, a synchronization is represented as a virtual data dependency edge, which is redundant when it does not carry actual data dependency. After removing all redundant synchronizations, SIRIUS performs code motion and post-fusion optimizations.

5.1 Removing redundant synchronizations

The key point for optimizing synchronizations is that for each data dependency edge from s to t, there should exist only one virtual data dependency edge from s to t. We abstract the problem of removing redundant synchronizations as a special case of barrier minimization problem, which has been well studied by a number of researchers (Hatcher & Quinn, 1991; O'Boyle & Stohr, 2002; Tseng, 1995). In particular, Darte and Schreiber proposed an algorithm to solve the problem in linear time (Darte & Schreiber, 2005). This paper implements a tailored version of the linear-time algorithm for our UAR (Algorithm 1).

Algorithm 1 traverses the fused UAR for each virtual data dependency edge, and removes it when it is redundant, where "redundant" is defined as follows.

- A virtual data dependency edge $e(s \to t)$ is redundant, if for any data dependency edge $e'(s' \to t')$, there does not exist a virtual data dependency path from s' to t' that contains e.
- A virtual data dependency edge $e(s \to t)$ is redundant, if for any data dependency edge $e'(s' \to t)$, all virtual data dependency paths from s' to t contain another virtual data dependency edge.
- A synchronization node is redundant if its out-degree is 0.

5.2 Post-fusion optimizations

Besides traditional compiler optimizations in LLVM, such as the redundant load elimination and dead code elimination discussed in the motivation example, SIRIUS extends the following post-fusion optimizations.

Code motion. After removing synchronizations, the fused UAR would be split into a set of connected components. There is no data dependency between these connected components, therefore SIRIUS applies code motion to enable them to share a common set of synchronizations. For example, the fragments of $(S1 \to P0 \to S2 \to S3)$ and $(S1' \to P0' \to S2')$ are changed into $(\{S1, S1'\} \to P0' \to \{S2, S2'\} \to S3)$.

Constant propagation. By uniformly representing the host and kernel codes in UAR, SIRIUS propagates the constant from the host to the kernel codes, thus the vendor compilers can generate optimized instructions using the constant values.

Reshaping iteration domains for mapping operations. When a fuse involves a fragment annotated as a pair-wise mapping or reduction operation, SIRIUS simply reuses the iteration domain of other fragments of the fuse and adjusts the generated code.

6 IMPLEMENTATION

We implement SIRIUS with about $7K$ lines of C++ code, including three major components: UAR construction, fusion derivation, and optimizing fused kernels.

We implement the component of UAR construction based on LLVM release/16.x. We use PipLib (PipLib) 1.3.3 as the ILP solver, which is in line with Pluto. After getting the schedule from the ISL solver, we write about $2K$ C++ codes to implement the functionality of generating fused kernel IR and arranging kernel fragments with the given fusion schedule.

7 EVALUATION

We have evaluated SIRIUS on six representative DNN models (ResNeXt (Xie et al., 2017), NASNet (Zoph et al., 2018), LSTM (Hochreiter & Schmidhuber, 1997), DeepSpeech2 (Amodei et al., 2016), Seq2Seq (Sutskever et al., 2014), and BERT (Devlin et al., 2018)). We focus our evaluation on model inference for DNN workloads.

LSTM, DeepSpeech2, Seq2Seq and NASNet are in line with Rammer (Ma et al., 2020). ResNext are trained on Cifar10. BERT is pre-trained on Wikipedia + Book Corpus Dataset. SIRIUS relies on CUDA source codes to construct UAR and perform optimization. The CUDA source code of SIRIUS contains 245 operators in total, of which 231 operators are from AutoTVM/Rammer, and 14 operators from BERT are manually implemented to get comparable performance with cuBLAS which is used by TensorRT. The trial number of AutoTVM (TVM v0.12) is set to 2000. We conduct the experiments on a server equipped with two Intel Xeon Gold 6248 CPUs, 768GB of DDR4 memory, and an NVIDIA A100 GPU. The server is installed with Ubuntu 22.04 and CUDA 11.8.

We compare the end-to-end performance with four DNN compilers: TensorFlow (TF, version 1.15.5), TensorRT(TRT, version 7.2), TVM, and Rammer (Ma et al., 2020). For the BERT network, we also compare it with AStitch (Zheng et al., 2022). The first 5 DNN models have the same hyper parameters as described in (Ma et al., 2020). BERT is a standard Bert-base model with the parameters (sequence length = 384, head num = layer num = 12, hidden dim = 64, float16). We also present the performance numbers on an NVIDIA Tesla V100 GPU as a reference. All performance numbers are averages over $1,000$ runs. We observe that there are little variances on the performance numbers.

7.1 Overall performances for DNN models

Shown by Figure 5, SIRIUS achieves average speedups of $41.78\times$/$21.76\times$/$19.16\times$/$5.15\times$(up to $154.84\times$/$60.80\times$/$43.69\times$/$11.98\times$) over TensorFlow/TF-XLA/TVM/TensorRT. Meanwhile, SIRIUS achieves an average of $1.62\times$ (upto $2.19\times$) over Rammer. The performance improvements come from two aspects. First, SIRIUS exploits more aggressive fusion opportunities thanks to whole-program analysis. Second, the enlarged kernel and vertical fusion open up more opportunities for sequential optimization (for a single thread).

7.2 Performance Breakdown

Figure 6 shows the performance breakdown of our key technologies for BERT and LSTM. For each one, the bar of `baseline` is the performance of the naive CUDA implementation, the bar of `+fusion` is the performance when enabling fusion in SIRIUS without supporting vertical fusion, the bar of `+h/v fusion` is the performance when coordinately considering horizontal and vertical fusion, the bar of `+seqopt` is the final performance when enabling sequential optimizations.

Take BERT for example, the results demonstrate that the simple fusion would reduce the execution time from 2.54ms (baseline) to 2.06ms due to the reduced cost of kernel launching. Then coordinately considering horizontal and vertical fusion does not change the performance. The reason is that the parallelism before horizontal fusion has been high enough to fully utilize the GPU SMs, thus arranging the code fragments horizontally or vertically would not change the execution. However, vertical fusion opens more opportunities for compiler optimizations and reduce the execution time to 1.79ms.

For LSTM, we can see similar observations. An exception is that when we change the horizontal fusion into vertical without optimization, the performance is decreased. The reason is that vertical fusion reduces the parallelism thus the GPU SMs cannot be fully utilized. But after the post-fusion optimization, the performance is improved and outperforms horizontal fusion.

We further take BERT as the example to discuss the effects of post-fusion optimizations. As shown in Figure 7, BASE denotes that the kernels are directly concatenated together with no optimizations applied, and its execution time is 2.06ms. Then SIRIUS removes redundant synchronizations and reduces the execution time to 2.04ms (OPTSYNC), and the optimization of code motion reduces the execution time to 1.86ms (CM). Finally the instruction optimizations reduces the execution time to 1.79ms (INSTOPT). Figure 7 also shows the effect of post-fusion optimizations on the number of instructions, with the right vertical axis.

7.3 Overhead Analysis

Our overhead comes from performing whole-program analysis, computing the fusion schedule using the ISL solver, applying post-fusion optimizations, and tuning for the code fragment arrangement (horizontal and vertical). Among these overhead, performing whole-program analysis and applying post-fusion optimizations consume less than 10 minutes for each DNN model. Computing the fusion schedule using the ISL solver takes less than 3 minutes, since for very large DNN models, such as NASNet and Bert, SIRIUS splits the UAR into small basic units and seeks for fusion schedule inside each basic unit. This scheme avoids to introduce extremely high cost for compiling the whole network, however, the basic unit has been large enough to enable the compiler to perform inter-procedure optimizations.

For very large DNN models, SIRIUS splits the UAR into a set of partitions according to user annotation and a threshold of the number of kernels. First, it enables users to annotate the building blocks in the network structure, such as the "cell" in NASNnet. Second, it sets a threshold of 100, i.e., when the number of kernels in the network exceeds the threshold, it would be split with the granularity of annotated building blocks. The building blocks in DNNs are typically loosely coupled, and the dependencies are concentrated inside each building block. Therefore, each partition is large enough to maintain the global views of one or more building blocks.

The overhead of tuning for horizontal and vertical fusion varies with DNN models, which depends on the number of nodes that can be executed in parallel. For BERT, SIRIUS needs to perform tuning among three candidates and it takes about 2 minutes, while for LSTM, SIRIUS searches through 545 candidates, finishing in 54 minutes, where NVCC spends most of the time.

SIRIUS takes the source code generated by TVM as inputs, and the long compilation time is a long-time known issue for TVM due to the large search space it formulates, e.g., compiling NASNet model requires 41.8 hours to finish 32% search progress according to Roller(Zhu et al., 2022). Thus, it is acceptable for SIRIUS to introduce extra <1 hour overhead.

	Runtime(ms)						Speedup over				
Network	TF	TF-XLA	TVM	Rammer	TRT	SIRIUS	TF	TF-XLA	TVM	Rammer	TRT
ResNext	41.61	123.42	47.78	3.86	24.32	2.03	20.50	60.80	23.54	1.90	11.98
NASNet	14.41	47.66	-	1.72	7.23	1.60	9.00	29.79	-	1.08	4.52
DeepSpeech2	38.28	11.53	20.87	3.09	3.92	1.59	24.11	7.26	13.15	1.94	2.47
LSTM	192.0	32.21	54.18	1.76	7.60	1.24	154.97	26.0	43.73	1.42	6.13
Seq2Seq	66.26	8.40	19.31	3.66	7.29	1.67	39.77	5.04	11.59	2.20	4.38
BERT	4.59	3.03	6.95	2.08	2.59	1.78	2.58	1.70	3.90	1.17	1.46

(a)

	Runtime(ms)						Speedup over				
Network	TF	TF-XLA	TVM	Rammer	TRT	SIRIUS	TF	TF-XLA	TVM	Rammer	TRT
ResNext	58.31	22.57	10.52	4.57	-	4.09	14.26	22.57	2.57	1.12	-
NASNet	23.51	20.57	2.81	2.01	4.16	1.89	12.44	20.57	1.48	1.06	2.20
DeepSpeech2	50.71	4.36	15.01	4.95	6.83	3.28	15.46	4.36	4.57	1.51	2.08
LSTM	153.40	20.54	30.54	3.82	10.59	2.98	51.48	20.54	10.24	1.28	3.55
Seq2Seq	97.67	11.27	23.23	3.87	30.68	1.94	50.35	11.27	11.97	1.99	15.81
BERT	7.80	5.88	-	7.10	3.76	3.02	2.58	1.97	-	2.35	1.25

(b)

Figure 5. End-to-end runtime and speedup for six DNN models on the NVIDIA A100 (a)/V100(b) system for batch size 1.

7.4 Case Study: The BERT model

SIRIUS speeds up the BERT model by $1.46\times$/$1.25\times$ over TensorRT on A100/V100 as shown in Figure 5. Furthermore, we also compare SIRIUS with AStitch (Zheng et al., 2022), Rammer (Ma et al., 2020) and TensorRT in this section. For BERT with batch size 1 on V100 platform, SIRIUS demonstrates the speedup of $1.61\times$, $2.35\times$, $1.25\times$ over AStitch, Rammer, TensorRT, respectively. We use the V100 platform since AStitch has not supported A100 yet.

Here we further discuss the performance benefit when comparing with AStitch, Rammer, and TensorRT. Consider the BERT sub-graph of 9 operators in Figure 1. AStitch is designed to fuse and optimize memory-intensive operators, and it can fuse the `add` and `reshape` operators, reducing kernel numbers to 6. Rammer is designed to fuse small kernels together to exploit inter-operator parallelism and fully utilize GPU SMs, For the sub-graph in Figure 1, it only fuses the three `add` operators together, leading to 7 kernels in total. TensorRT manually fuses the three `GEMM` and three `add` operators into one kernel while leaving `reshape` operators not fused. This results in a total of 4 kernels. In comparison, SIRIUS has only 1 kernel after fusion.

Impacts of batch size. We further evaluate SIRIUS when varying the batch size (i.e., 1, 4, 16). Figure 8 shows their performance normalized to TensorFlow. SIRIUS achieves an average speedup of $1.56\times$ over AStitch, $2.92\times$ over Rammer, and $1.48\times$ over TensorRT. In summary, SIRIUS can improve BERT performance for different batch sizes.

7.5 Case Study: The LSTM model

For LSTM, SIRIUS can obtain significant performance improvement, i.e, $6.13\times$ over TensorRT and $1.42\times$ over Rammer. Figure 9(a) shows the code skeleton of LSTM and (b) shows the data dependencies. Each dot in (b) represents a LSTM cell with each one including 8 `gemv` invocations and 1 `solve` invocation. It shows that there does not exist data dependencies along the back-diagonal dimension, both SIRIUS and Rammer can finds the fusion opportunity along the back-diagonal dimension, which is called as wavefront in Rammer.

For LSTM, SIRIUS and Rammer show two differences. First, SIRIUS fuses all the `gemv` and `solve` from all the cells along the wavefront dimension, and gets one kernel for each wave, while Rammer generates two kernels for each wave, one for `gemv` and the other for `solve`. Second, the code fragments are arranged in a different way, i.e., horizontal in Rammer and vertical in SIRIUS, as shown in Figure 9(c). The enlarged fused kernel and vertical arrangement enables more instruction optimizations, as discussed below.

In comparison, TensorRT only fuses the eight gemv kernels into one but fails to perform the skewed fusions. It launches 1040 gemv and 1000 solve kernels for the single batch case. In addition, the fused kernel still generates MAD instruc-

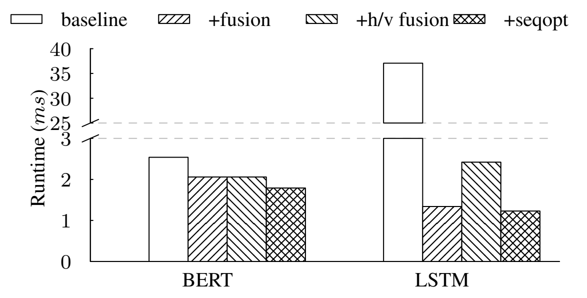


Figure 6. Performance breakdown of SIRIUS.

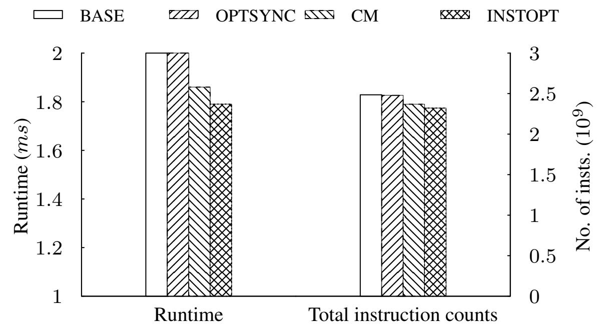


Figure 7. Effects of post-fuse optimizations.

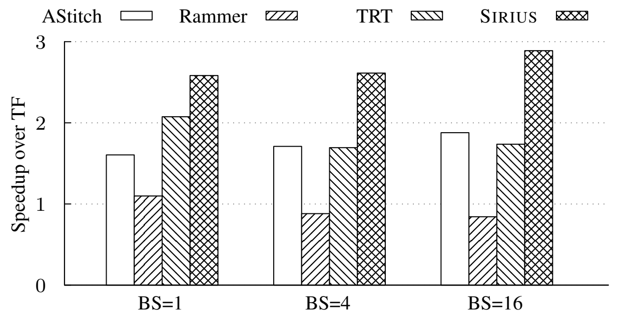
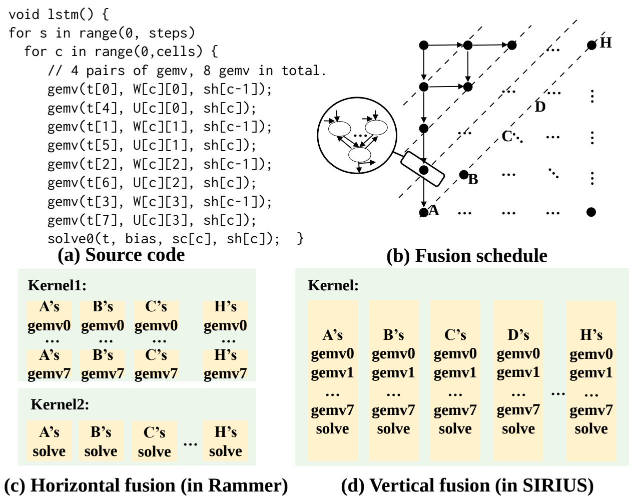


Figure 8. Performance results for varying batch sizes on V100.

tions to compute the addresses for memory loads, resulting $4.51\times$ more instruction counts compared to the one from SIRIUS.

Instruction optimizations. Figure 10 describes various performance characteristics for three different implementations of a cell in the LSTM network. Analysis shows that there are two to three times differences on the total number of instructions executed. The baseline includes four kernels for the multiplication and four kernels for fused multiplication and partial-aggregation. Rammer fuses the eight `gemv` kernels into one single kernel and generates one kernel for the aggregation `solve`. One thing worth noting is that for each FMA instruction, both implementations from Baseline and Rammer need to issue a MAD instruction to compute the address of the matrix element before issuing a load instruction. SIRIUS propagates the fact that the host allocates the eight matrices in a contiguous memory region to the fused kernel. It rewrites the base addresses of the eight matrices as an offset to a common base address, and groups the multiplications within the same basic block. The compiler from the vendors (i.e., NVCC (NVIDIA Corporation., 2021)) is able to compute the offset in compile-time and to embed them directly into the load instructions. The end result is that the kernel from SIRIUS issues 1.56 auxiliary instructions per FMA instruction per iteration.



```
void lstm() {
for s in range(0, steps)
  for c in range(0,cells) {
    // 4 pairs of gemv, 8 gemv in total.
    gemv(t[0], W[c][0], sh[c-1]);
    gemv(t[4], U[c][0], sh[c]);
    gemv(t[1], W[c][1], sh[c-1]);
    gemv(t[5], U[c][1], sh[c]);
    gemv(t[2], W[c][2], sh[c-1]);
    gemv(t[6], U[c][2], sh[c]);
    gemv(t[3], W[c][3], sh[c-1]);
    gemv(t[7], U[c][3], sh[c]);
    solve0(t, bias, sc[c], sh[c]);  }
```

(a) Source code
H
D
C
B
A
(b) Fusion schedule
Kernel1:
A's gemv0 … A's gemv7
B's gemv0 … B's gemv7
C's gemv0 … C's gemv7
H's gemv0 … H's gemv7
Kernel2:
A's solve
B's solve
C's solve
H's solve
Kernel:
A's gemv0 gemv1 … gemv7 solve
B's gemv0 gemv1 … gemv7 solve
C's gemv0 gemv1 … gemv7 solve
D's gemv0 gemv1 … gemv7 solve
H's gemv0 gemv1 … gemv7 solve
(c) Horizontal fusion (in Rammer)
(d) Vertical fusion (in SIRIUS)

Figure 9. Fusion schedule for LSTM in SIRIUS.

Implementation	Baseline	Rammer	SIRIUS
# of Kernels	9	2	1
# of Threads	18432	8448	2048
# of FMA inst.	528,200	531,832	528,200
# of aux. insts. / FMA / iterations	6.1	3.1	1.9
Total inst.	4,300,106	2,723,558	1,669,450
Runtime (μs)	58.1	12.8	9.1

(a)

```
IMAD.WIDE R14, R17, R10,
  c[0x0][0x160];
IMAD.WIDE R16, R19,
R10.reuse,
  c[0x0][0x170];
LDG.E R14, [R14.64];
LDG.E R17, [R16.64];
IMAD.WIDE R18, R19, R10,
  c[0x0][0x160];
LDG.E R18, [R18.64];
```

(b)

```
LDG.E R23,
[R4.64+-0x140c00];
LDG.E R21, [R60.64+-0x8];
LDG.E R20,
[R4.64+-0x1c0c00];
......
FFMA R22, R21, R22, R14;
FFMA R14, R21.reuse, R23,
R15;
```

(c)

Figure 10. (a) Performance characteristics of the implementation of the LSTM cell kernel from Baseline, Rammer and SIRIUS on an NVIDIA A100 system. (b) / (c) The assembly code snippets for Rammer and SIRIUS.

8 RELATED WORK

Optimizing accelerator programs. When developing DNN applications, users specify the network as a series of tensor transformations or a computation graph whose nodes represent a DNN operator using frameworks like TensorFlow (Abadi et al., 2016), TVM (Chen et al., 2018a), Tensor Comprehension (Vasilache et al., 2019). Previous work optimizes DNN implementations through pattern substitutions and kernel fusions (Chen et al., 2018b; Jia et al., 2019a; Leary & Wang, 2017; Niu et al., 2021; Truong et al., 2016; Zheng et al., 2020a;c), systematic scheduling to utilize parallelisms (Gao et al., 2019; Ma et al., 2020; Oh et al., 2021), and auto tuning (Chen et al., 2018a; Zhao et al., 2019b; Zheng et al., 2020b). Similar issues arise on image processings and stencil computation kernels. Halide (Ragan-Kelley et al., 2013) and Pencil (Baghdadi et al., 2015) enables users to specify the applications in domain-specific languages (DSL). Tiramisu (Baghdadi et al., 2019) and Fireiron (Hagedorn et al., 2020) develop DSL to specify the schedules and memory layouts independently from the computations to match the characteristics of the hardware.

Optimizing synchronization on parallel accelerators is crucial for accelerator programs. A typical problem is the barrier minimization problem, which has been extensively explored (Darte & Schreiber, 2005; Hatcher & Quinn, 1991; O'Boyle & Stohr, 2002; Tseng, 1995). (Hatcher & Quinn, 1991) presents a formalization of the barrier minimization problem and suggests algorithms to minimize barriers within nested inner loops. (O'Boyle & Stohr, 2002) proposes an algorithm to provably place the minimum number of barriers for perfect loop nests and certain imperfect loop nest sequences. (Darte & Schreiber, 2005) proposes a linear-time algorithm to solve this problem. These works provide valuable references for post-fusion optimization in SIRIUS.

Polyhedral compilers. Polyhedral analysis (Baghdadi et al., 2019; Bondhugula et al., 2008; Grosser et al., 2012; Hartono et al., 2009; Mullapudi et al., 2015) models nested loops as polyhedra and reasons about the transformations on top of the polyhedra. Polyhedral analysis is able to drive complex transformations like loop interchanges (Allen & Kennedy, 1984), tiling (Hartono et al., 2009; Zhao & Di, 2020), and loop fusions (Bondhugula et al., 2008). Research has shown that it is effective on optimizing scientific applications, image processing and DNNs (Baghdadi et al., 2019; Cowan et al., 2020; Pradelle et al., 2019; Zhao et al., 2021). Research on improving the generality (Benabderrahmane et al., 2010) and scalability (Vasilache et al., 2006) of polyhedral compilers broadens the applicability of the methods. (Turner et al., 2021) presents a new unified program transformation approach with the polyhedral model to optimizing convolutional neural networks, further reducing neural architecture search (NAS) search time. SIRIUS combines the ideas of inter-procedural analysis (Lattner et al., 2007) and polyhedral representation to capture a fine-grain picture of the dependency of the full program. The fine-grain picture allows SIRIUS to drive optimizations across codes on both the host side and the device side.

Compiler supports for innovative architectures for accelerators. Innovative architectures (Jia et al., 2019b; Jouppi et al., 2017; Liao et al., 2019; Rocki et al., 2020; Zhao et al., 2019a) presents unique challenges for compilers due to their multi-dimensional memory hierarchies, trade-offs between parallelisms and locality, etc. AKG (Zhao et al., 2021) demonstrates that the techniques of polyhedral compilers. SIRIUS is orthogonal as it focuses on generating fusion schedules that generate efficient sequential code for the accelerators. It is possible to extend SIRIUS on these architectures to further improve the performances of the applications.

Closest related work. Recently, DNNFusion (Niu et al., 2021) proposed to use a set of rules instead of patterns for operator fusion, in comparison, SIRIUS further fuses multiple independent branches together. AStitch (Zheng et al., 2022) proposed to fuse memory-intensive operators, in comparison, SIRIUS further fuses memory-intensive operators and computation-intensive operators (maybe libraries). Rammer (Ma et al., 2020) proposed to fuse small kernels for parallelism stitching, in comparison, SIRIUS can fuse large and small kernel together, and further exploit vertical fusion. Apollo (Zhao et al., 2022) leverages partition-based approach to first split the graph into sub-graphs, and then search for fusion opportunities. However, Apollo uses a set of rules for the graph splitting, in comparison, SIRIUS leverages compiler dependency analysis to determine the fusion schedule. For the subgraph in Figure 1, only SIRIUS fuses the 9 operators into one kernel.

9 CONCLUSION AND FUTURE WORK

This paper proposes an optimizing compiler framework, SIRIUS. By modeling the host and kernel codes in a unified polyhedral representation, SIRIUS leverages polyhedral analysis to expose maximal kernel fusion opportunities then generates the fused kernels. Finally the fused kernel can benefit from many traditional sequential optimizations. Evaluations demonstrate that SIRIUS can achieve up to $11.98\times$ speedup over TensorRT. Specifically, for BERT, SIRIUS can achieve $1.46\times$ speedup over TensorRT.

SIRIUS suffers from two limitations. First, it requires to see the CUDA source codes of the operators. Now SIRIUS uses the CUDA codes generated by TVM. Our future work will take the tensor expressions to facilitate the analysis and optimizations. Second, SIRIUS cannot change the thread organizations since it is an optimizing compiler after CUDA code generation. Our future work will hoist it to TVM so that the thread organizations can be optimized globally. And we will leverage the SIRIUS approach to support the training process in the future.

ACKNOWLEDGEMENTS

We thank anonymous reviewers and our shepherd, Prof. Michael O'Boyle, for their extensive suggestions. This work was partially supported by the National Key R&D Program of China (2021ZD0110101), the National Natural Science Foundation of China (62232015, 62090024, U20A20226).

REFERENCES

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283, Savannah, GA, November 2016. USENIX Association. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

Ahmed, I., Parmar, S., Boyd, M., Beidler, M., Kang, K., Liu, B., Roach, K., Kim, J., and Abts, D. Answer fast: Accelerating bert on the tensor streaming processor, 2022.

Allen, J. R. and Kennedy, K. Automatic loop interchange. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '84, pp. 233–246, New York, NY, USA, 1984. Association for Computing Machinery. ISBN 0897911393. doi: 10.1145/502874.502897. URL https://doi.org/10.1145/502874.502897.

Amodei, D., Ananthanarayanan, S., Anubhai, R., Bai, J., Battenberg, E., Case, C., Casper, J., Catanzaro, B., Cheng, Q., Chen, G., et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International conference on machine learning*, pp. 173–182. PMLR, 2016.

Andersen, L. O. *Program analysis and specialization for the C programming language*. PhD thesis, DIKU, University of Copenhagen, 1994.

Baghdadi, R., Beaugnon, U., Cohen, A., Grosser, T., Kruse, M., Reddy, C., Verdoolaege, S., Betts, A., Donaldson, A. F., Ketema, J., Absar, J., Van Haastregt, S., Kravets, A., Lokhmotov, A., David, R., and Hajiyev, E. Pencil: A platform-neutral compute intermediate language for accelerator programming. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pp. 138–149, 2015. doi: 10.1109/PACT.2015.17.

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

Benabderrahmane, M.-W., Pouchet, L.-N., Cohen, A., and Bastoul, C. The polyhedral model is more widely applicable than you think. In Gupta, R. (ed.), *Compiler Construction*, pp. 283–303, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-11970-5.

Bondhugula, U., Hartono, A., Ramanujam, J., and Sadayappan, P. A practical automatic polyhedral program optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2008.

Bondhugula, U. K. *Effective automatic parallelization and locality optimization using the polyhedral model*. PhD thesis, The Ohio State University, 2008.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H. (eds.), *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578–594, Carlsbad, CA, October 2018a. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/chen.

Chen, T., Zheng, L., Yan, E., Jiang, Z., Moreau, T., Ceze, L., Guestrin, C., and Krishnamurthy, A. Learning to optimize tensor programs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, pp. 3393–3404, Red Hook, NY, USA, 2018b. Curran Associates Inc.

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

Darte, A. and Schreiber, R. A linear-time algorithm for optimal barrier placement. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pp. 26–35, 2005.

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

Fu, C., Zhu, S., Chen, H., Koushanfar, F., Su, H., and Zhao, J. Simbnn: A similarity-aware binarized neural network acceleration framework. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 319–319. IEEE, 2019.

Gao, M., Yang, X., Pu, J., Horowitz, M., and Kozyrakis, C. Tangram: Optimized coarse-grained dataflow for scalable nn accelerators. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 807–820, 2019.

González-Alvarez, C., Sartor, J. B., Álvarez, C., Jiménez-González, D., and Eeckhout, L. Mingle: an efficient framework for domain acceleration using low-power specialized functional units. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(2):1–26, 2016.

Grosser, T., Groesslinger, A., and Lengauer, C. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22 (04), 2012.

Gupta, U., Hsia, S., Saraph, V., Wang, X., Reagen, B., Wei, G.-Y., Lee, H.-H. S., Brooks, D., and Wu, C.-J. Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 982–995. IEEE, 2020.

Hagedorn, B., Elliott, A. S., Barthels, H., Bodik, R., and Grover, V. Fireiron: A data-movement-aware scheduling language for gpus. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, PACT '20, pp. 71–82, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450380751. doi: 10.1145/3410463.3414632. URL https://doi.org/10.1145/3410463.3414632.

Harris, M. et al. Optimizing parallel reduction in cuda. *Nvidia developer technology*, 2(4):70, 2007.

Hartono, A., Baskaran, M. M., Bastoul, C., Cohen, A., Krishnamoorthy, S., Norris, B., Ramanujam, J., and Sadayappan, P. Parametric multi-level tiling of imperfectly nested loops. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pp. 147–157, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605584980. doi: 10.1145/1542275.1542301. URL https://doi.org/10.1145/1542275.1542301.

Hatcher, P. J. and Quinn, M. J. *Data-parallel programming on MIMD computers*, volume 90. Mit Press, 1991.

Hemmat, M., Shah, T., Chen, Y., and San Miguel, J. Crania: Unlocking data and value reuse in iterative neural network architectures. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 295–300. IEEE, 2020.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL https://doi.org/10.1162/neco.1997.9.8.1735.

Jia, W., Wang, H., Chen, M., Lu, D., Lin, L., Car, R., Weinan, E., and Zhang, L. Pushing the limit of molecular dynamics with ab initio accuracy to 100 million atoms with machine learning. In *SC20: International conference for high performance computing, networking, storage and analysis*, pp. 1–14. IEEE, 2020.

Jia, Z., Padon, O., Thomas, J., Warszawski, T., Zaharia, M., and Aiken, A. Taso: Optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pp. 47–62, New York, NY, USA, 2019a. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359630. URL https://doi.org/10.1145/3341301.3359630.

Jia, Z., Tillman, B., Maggioni, M., and Scarpazza, D. P. Dissecting the graphcore ipu architecture via microbenchmarking, 2019b.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P.-l., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pp. 1–12, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450348928. doi: 10.1145/3079856.3080246. URL https://doi.org/10.1145/3079856.3080246.

Judd, P., Albericio, J., Hetherington, T., Aamodt, T. M., Jerger, N. E., and Moshovos, A. Proteus: Exploiting numerical precision variability in deep neural networks. In *Proceedings of the 2016 International Conference on Supercomputing*, pp. 1–12, 2016.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017. ISSN 0001-0782. doi: 10.1145/3065386. URL https://doi.org/10.1145/3065386.

Kwon, H., Chatarasi, P., Pellauer, M., Parashar, A., Sarkar, V., and Krishna, T. Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 754–768, 2019.

Lattner, C. and Adve, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pp. 75–86. IEEE, 2004.

Lattner, C., Lenharth, A., and Adve, V. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pp. 278–289, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595936332. doi: 10.1145/1250734.1250766. URL https://doi.org/10.1145/1250734.1250766.

Leary, C. and Wang, T. XLA: Tensorflow, compiled. TesorFlow Dev Summit, 2017.

Liao, H., Tu, J., Xia, J., and Zhou, X. Davinci: A scalable architecture for neural network computing. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pp. 1–44, 2019. doi: 10.1109/HOTCHIPS.2019.8875654.

Liu, A., Bernstein, G. L., Chlipala, A., and Ragan-Kelley, J. Verified tensor-program optimization via high-level scheduling rewrites. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–28, 2022.

Ma, L., Xie, Z., Yang, Z., Xue, J., Miao, Y., Cui, W., Hu, W., Yang, F., Zhang, L., and Zhou, L. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 881–897. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/ma.

Mullapudi, R. T., Vasista, V., and Bondhugula, U. Polymage: Automatic optimization for image processing pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pp. 429–443, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450328357. doi: 10.1145/2694344.2694364. URL https://doi.org/10.1145/2694344.2694364.

Ning, L., Guan, H., and Shen, X. Adaptive deep reuse: Accelerating cnn training on the fly. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pp. 1538–1549. IEEE, 2019.

Niu, W., Guan, J., Wang, Y., Agrawal, G., and Ren, B. 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*, PLDI 2021, pp. 883–898, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383912. doi: 10.1145/3453483.3454083. URL https://doi.org/10.1145/3453483.3454083.

NVIDIA Corporation. Tensorrt. https://developer.nvidia.com/tensorrt, a.

NVIDIA Corporation. Basic linear algebra on nvidia gpus. https://developer.nvidia.com/cublas, b.

NVIDIA Corporation. Nvidia tesla v100 gpu architecture. https://www.nvidia.com/en-us/data-center/volta-gpu-architecture/, 2017.

NVIDIA Corporation. Cuda toolkit — nvidia developer. https://developer.nvidia.com/cuda-toolkit, 2021.

O'Boyle, M. and Stohr, E. Compile time barrier synchronization minimization. *IEEE Transactions on Parallel and Distributed Systems*, 13(6):529–543, 2002.

Oh, Y. H., Kim, S., Jin, Y., Son, S., Bae, J., Lee, J., Park, Y., Kim, D. U., Ham, T. J., and Lee, J. W. Layerweaver: Maximizing resource utilization of neural processing units via layer-wise scheduling. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 584–597. IEEE, 2021.

Pearce, D. J., Kelly, P. H., and Hankin, C. Efficient field-sensitive pointer analysis of c. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(1): 4–es, 2007.

PipLib. Pip: The parametric integer programming library. http://www.piplib.org.

Pradelle, B., Meister, B., Baskaran, M., Springer, J., and Lethin, R. Polyhedral optimization of tensorflow computation graphs. In Bhatele, A., Boehme, D., Levine, J. A., Malony, A. D., and Schulz, M. (eds.), *Programming and Performance Visualization Tools*, pp. 74–89, Cham, 2019. Springer International Publishing. ISBN 978-3-030-17872-7.

Qin, E., Samajdar, A., Kwon, H., Nadella, V., Srinivasan, S., Das, D., Kaul, B., and Krishna, T. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 58–70. IEEE, 2020.

Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. 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*, PLDI '13, pp. 519–530, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320146. doi: 10.1145/2491956.2462176. URL https://doi.org/10.1145/2491956.2462176.

Rocki, K., Van Essendelft, D., Sharapov, I., Schreiber, R., Morrison, M., Kibardin, V., Portnoy, A., Dietiker, J. F., Syamlal, M., and James, M. Fast stencil-code computation on a wafer-scale processor. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14. IEEE, 2020.

Sutskever, I., Vinyals, O., and Le, Q. V. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, pp. 3104–3112, Cambridge, MA, USA, 2014. MIT Press.

Truong, L., Barik, R., Totoni, E., Liu, H., Markley, C., Fox, A., and Shpeisman, T. Latte: A language, compiler, and runtime for elegant and efficient deep neural networks. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pp. 209–223, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342612. doi: 10.1145/2908080.2908105. URL https://doi.org/10.1145/2908080.2908105.

Tseng, C.-W. Compiler optimizations for eliminating barrier synchronization. *ACM SIGPLAN Notices*, 30(8):144–155, 1995.

Turner, J., Crowley, E. J., and O'Boyle, M. F. Neural architecture search as program transformation exploration. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 915–927, 2021.

Vasilache, N., Bastoul, C., and Cohen, A. Polyhedral code generation in the real world. In Mycroft, A. and Zeller, A. (eds.), *Compiler Construction*, pp. 185–201, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-33051-6.

Vasilache, N., Zinenko, O., Theodoridis, T., Goyal, P., Devito, Z., Moses, W. S., Verdoolaege, S., Adams, A., and Cohen, A. The next 700 accelerated layers: From mathematical expressions of network computation graphs to accelerated gpu kernels, automatically. *ACM Trans. Archit. Code Optim.*, 16(4), October 2019. ISSN 1544-3566. doi: 10.1145/3355606. URL https://doi.org/10.1145/3355606.

Verdoolaege, S., Janssens, G., and Leuven, K. Scheduling for ppcg. In *CW ReportsCW Reports*, 2017.

Wahib, M. and Maruyama, N. Scalable kernel fusion for memory-bound gpu applications. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 191–202, 2014. doi: 10.1109/SC.2014.21.

Wang, S., Yang, P., Zheng, Y., Li, X., and Pekhimenko, G. Horizontally fused training array: An effective hardware utilization squeezer for training novel deep learning models. *Proceedings of Machine Learning and Systems*, 3: 599–623, 2021.

Whaley, J. and Lam, M. S. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming Language Design and Implementation*, pp. 131–144, 2004.

Xiao, S. and Feng, W.-c. Inter-block gpu communication via fast barrier synchronization. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pp. 1–12. IEEE, 2010.

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

Zhang, Q., Han, Z., Yang, F., Zhang, Y., Liu, Z., Yang, M., and Zhou, L. Retiarii: A deep learning Exploratory-Training framework. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 919–936. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/zhang-quanlu.

Zhao, J. and Di, P. Optimizing the memory hierarchy by compositing automatic transformations on computations and data. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 427–441, 2020. doi: 10.1109/MICRO50266.2020.00044.

Zhao, J., Li, B., Nie, W., Geng, Z., Zhang, R., Gao, X., Cheng, B., Wu, C., Cheng, Y., Li, Z., Di, P., Zhang, K., and Jin, X. 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*, PLDI 2021, pp. 1233–1248, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383912. doi: 10.1145/3453483.3454106. URL https://doi.org/10.1145/3453483.3454106.

Zhao, J., Gao, X., Xia, R., Zhang, Z., Chen, D., Chen, L., Zhang, R., Geng, Z., Cheng, B., and Jin, X. Apollo: Automatic partition-based operator fusion through layer by layer optimization. mlsys 2022. In *MLSys*, 2022.

Zhao, Y., Du, Z., Guo, Q., Liu, S., Li, L., Xu, Z., Chen, T., and Chen, Y. Cambricon-f: Machine learning computers with fractal von neumann architecture. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pp. 788–801, New York, NY, USA, 2019a. Association for Computing Machinery. ISBN 9781450366694. doi: 10.1145/3307650.3322226. URL https://doi.org/10.1145/3307650.3322226.

Zhao, Z., Kwon, H., Kuhar, S., Sheng, W., Mao, Z., and Krishna, T. mrna: Enabling efficient mapping space exploration for a reconfiguration neural accelerator. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 282–292. IEEE, 2019b.

Zheng, B., Vijaykumar, N., and Pekhimenko, G. Echo: Compiler-based gpu memory footprint reduction for lstm rnn training. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, pp. 1089–1102. IEEE Press, 2020a. ISBN 9781728146614. doi: 10.1109/ISCA45697.2020.00092. URL https://doi.org/10.1109/ISCA45697.2020.00092.

Zheng, L., Jia, C., Sun, M., Wu, Z., Yu, C. H., Haj-Ali, A., Wang, Y., Yang, J., Zhuo, D., Sen, K., Gonzalez, J. E., and Stoica, I. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 863–879. USENIX Association, November 2020b. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/zheng.

Zheng, S., Liang, Y., Wang, S., Chen, R., and Sheng, K. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pp. 859–873, New York, NY, USA, 2020c. Association for Computing Machinery. ISBN 9781450371025. doi: 10.1145/3373376.3378508. URL https://doi.org/10.1145/3373376.3378508.

Zheng, Z., Yang, X., Zhao, P., Long, G., Zhu, K., Zhu, F., Zhao, W., Liu, X., Yang, J., Zhai, J., et al. Astitch: enabling a new multi-dimensional optimization space for memory-intensive ml training and inference on modern simt architectures. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 359–373, 2022.

Zhu, H., Wu, R., Diao, Y., Ke, S., Li, H., Zhang, C., Xue, J., Ma, L., Xia, Y., Cui, W., et al. {ROLLER}: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 233–248, 2022.

Zinenko, O., Verdoolaege, S., Reddy, C., Shirako, J., Grosser, T., Sarkar, V., and Cohen, A. Unified polyhedral modeling of temporal and spatial locality. In *Research Report RR-9110, Inria Paris*, 2017.

Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. Learning transferable architectures for scalable image recognition. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pp. 8697–8710. IEEE Computer Society, 2018. doi: 10.1109/CVPR.2018.00907. URL http://openaccess.thecvf.com/content_cvpr_2018/html/Zoph_Learning_Transferable_Architectures_CVPR_2018_paper.html.

A ARTIFACT APPENDIX

A.1 Abstract

The artifact contains the necessary software components to validate the main results in this paper. We provide a docker image to ease the environment setup. The docker image contains the source code of Sirius, scripts to evaluate the inference performance, and scripts to draw the figures. It requires a Linux system with NVIDIA driver (capable to run CUDA 11.8) running on a NVIDIA A100 Tensor Core GPU equipped x86_64 machine to create the docker container. After launching the docker container, you can run scripts to collect performance numbers and draw pictures.

A.2 Artifact check-list (meta-information)

- **Model: ResNext, NANet, DeepSpeech2, LSTM, Seq2Seq and**

  BERT(included in docker image)
- **Run-time environment: A Linux system with NVIDIA driver(capable to run CUDA 11.8)**
- **Hardware: NVIDIA A100 Tensor Core GPU**
- **Metrics: End-to-end inference time**
- **Output: Key graphs and necessary data**
- **Experiments: See read-me file in docker image.**
- **How much disk space required (approximately)?: 60GB**
- **How much time is needed to prepare workflow (approximately)?: 20 minutes**
- **How much time is needed to complete experiments (approximately)?: It requires dozens of minutes to download the docker image. You can then run a script once to collect all performance results and draw pictures. It requires about 2 hours in total.**
- **Publicly available?: Yes. The docker image is public, which contains the source code.**
- **Code licenses (if publicly available)?: The GNU General Public License (GPL)**
- **Workflow framework used?: LLVM**
- **Archived (provide DOI)?: https://zenodo.org/record/7885573**

A.3 Description

A.3.1 How to access

We provide the docker image at both dockerhub and zenodo.

Docker-hub URL: https://hub.docker.com/r/sunqianqi/sirius

Zenodo URL: https://zenodo.org/record/7885573

A.3.2 Hardware dependencies

NVIDIA A100 Tensor Core GPU equipped x86_64 machines.

A.3.3 Software dependencies

- Linux system with NVIDIA driver capable to run CUDA 11.8
- Docker version 20.10.14

A.3.4 Models

Required Models have been included in the docker, do not need extra work. Models include ResNext, NANet, DeepSpeech2, LSTM, Seq2Seq and BERT.

A.4 Installation

You just need to pull the docker image and launch a container:

```
$ docker pull sunqianqi/sirius:mlsys_ae
$ docker run -it --name=sirius_test \
  --gpus all --privileged \
  sunqianqi/sirius:mlsys_ae /bin/bash
```

Use sudo to run docker if necessary.

A.5 Experiment workflow

A.6 Evaluation and expected results

We have provided a README file (**/root/mlsys_ae/README.md**) on how to reproduce the results within our provided docker image.

You can reproduce experiment results in SIRIUS paper following the steps in README.md.