Exocompilation for productive programming of hardware accelerators

Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. 2022.

차세대 컴퓨터 시스템 연구실

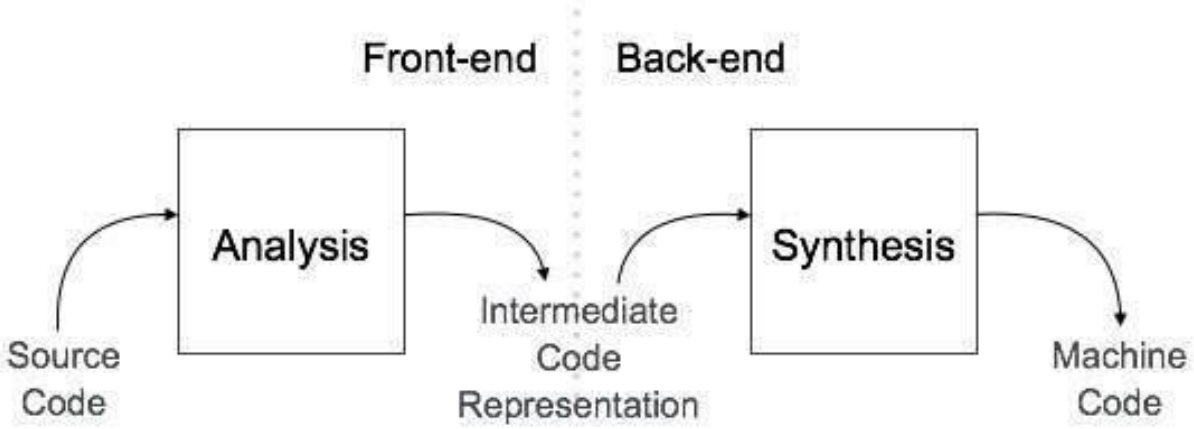
이원호

Compiler

- IR (Intermediate Representation)

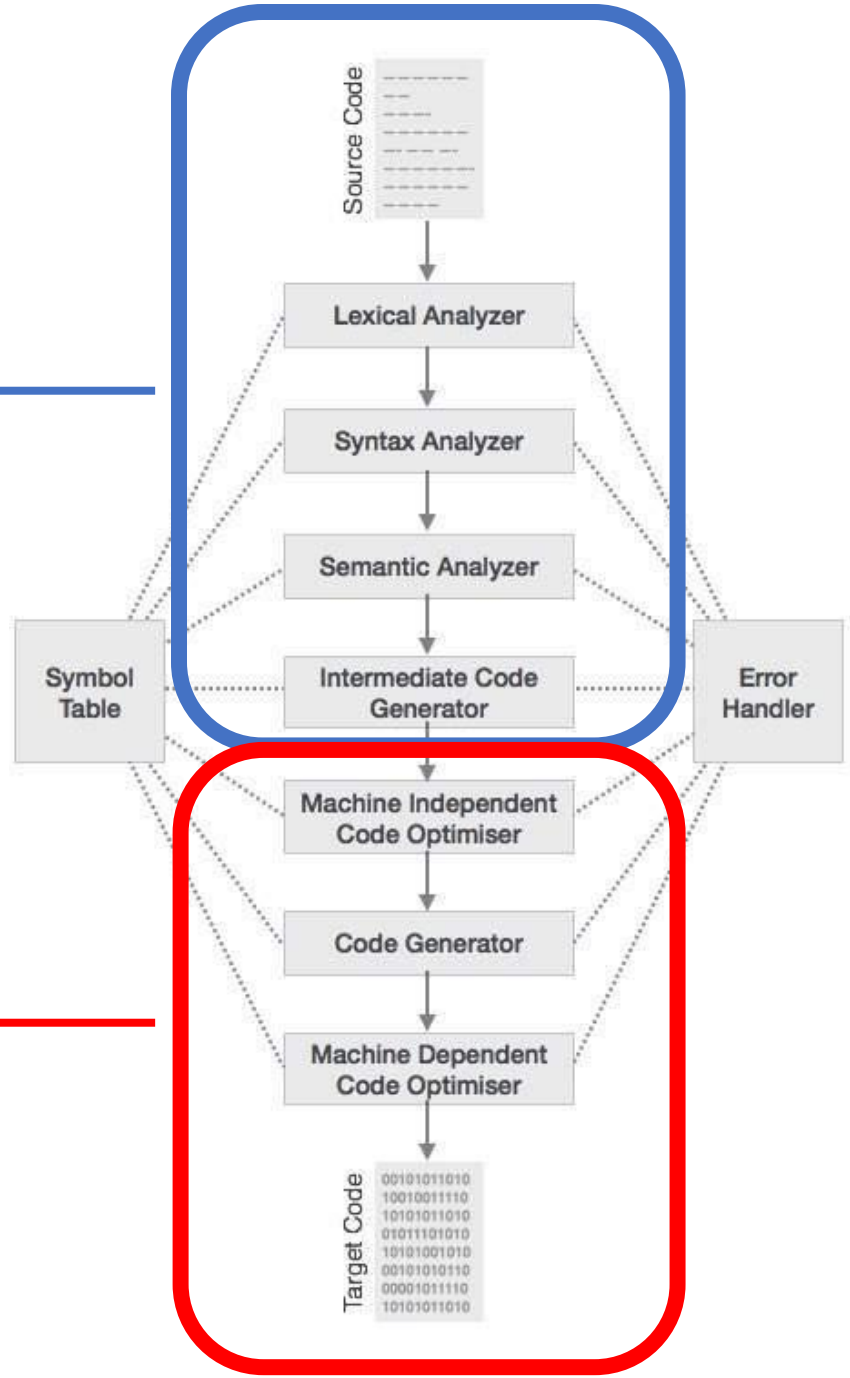
- Front-end
  - Source code → IR

- Back-end
  - IR → Machine code

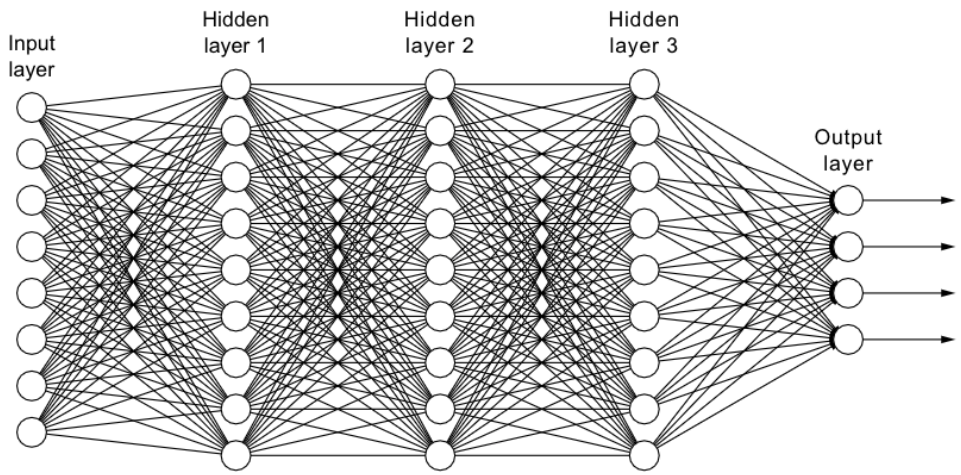


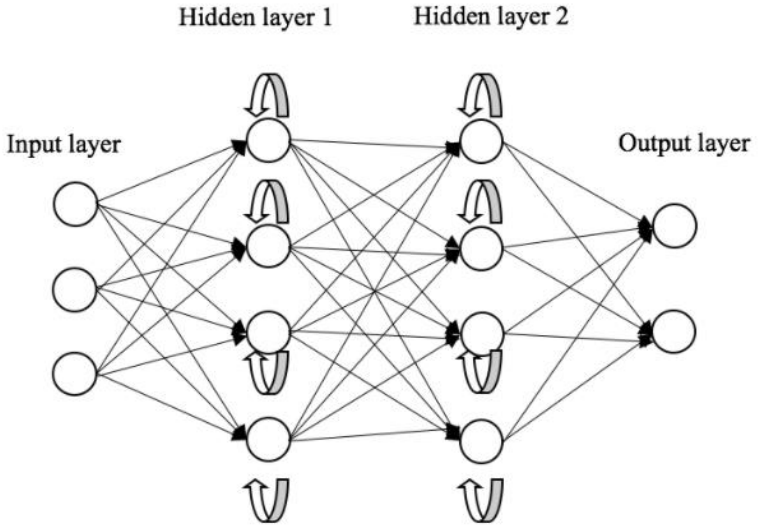
Compiler

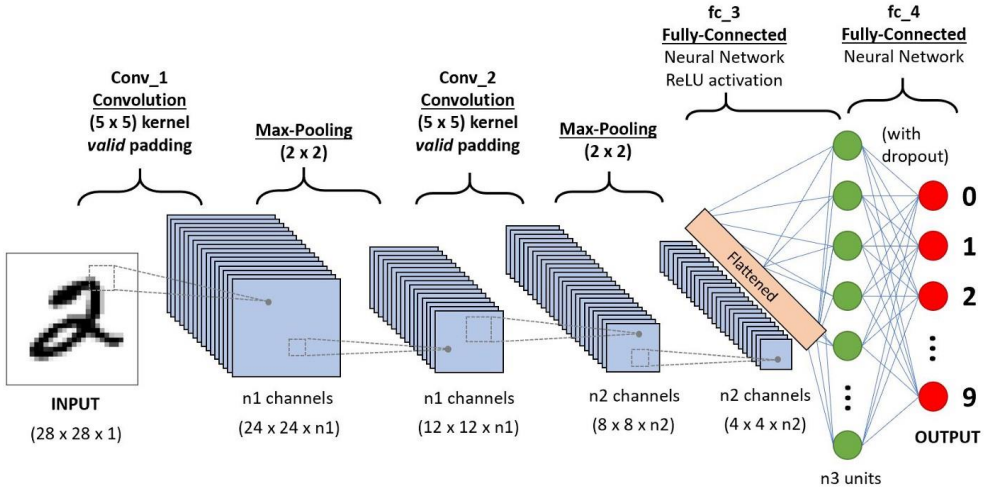
- Front-end (source code → IR)
  - Lexical Analysis
  - Syntax Analysis
  - Semantic Analysis
  - Intermediate Code Generator
- Back-end (IR → machine code)
  - Code Optimizing
  - Code Generator

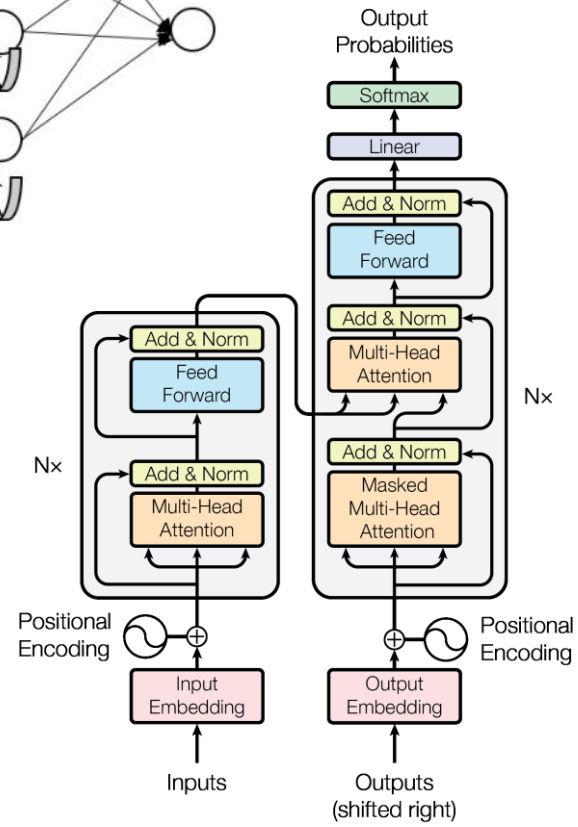


Deep Learning



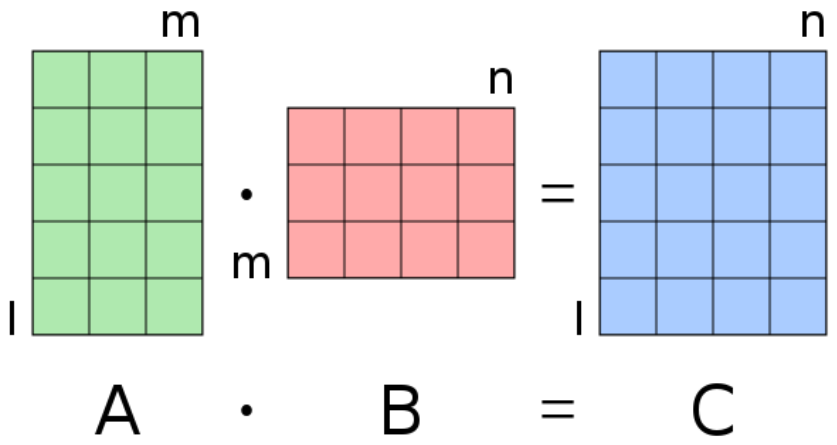




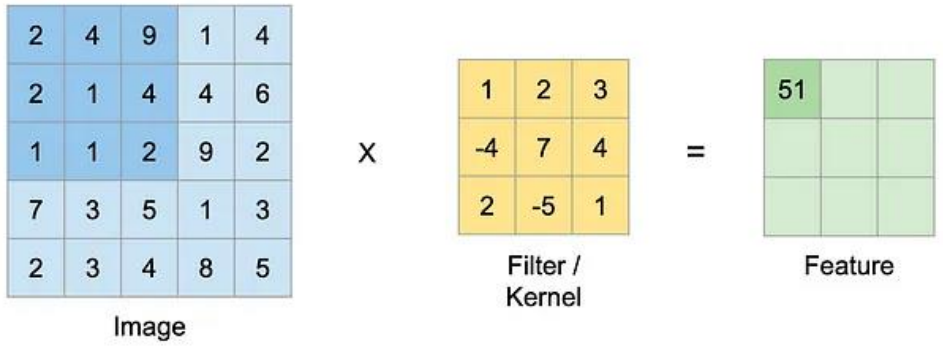


Deep Learning

- Two Basic Operation
  - GEMM (General Matrix Multiplication)



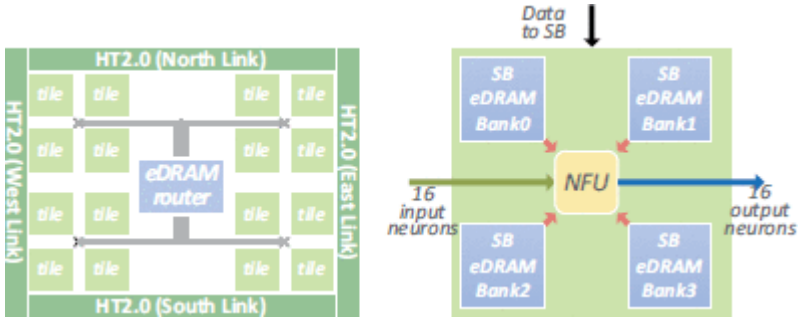
  - CONV (Convolution)



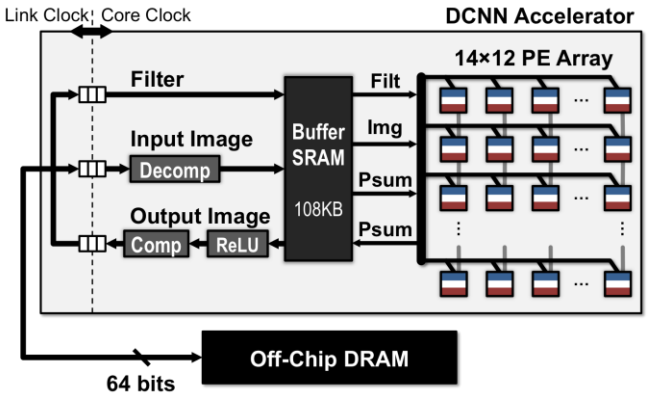
DL Accelerator

- Different Architecture has different ISA
  - Special instructions for Deep Learning
- Three Key points
  - Parallelism
  - Dataflow
  - Memory Hierarchy

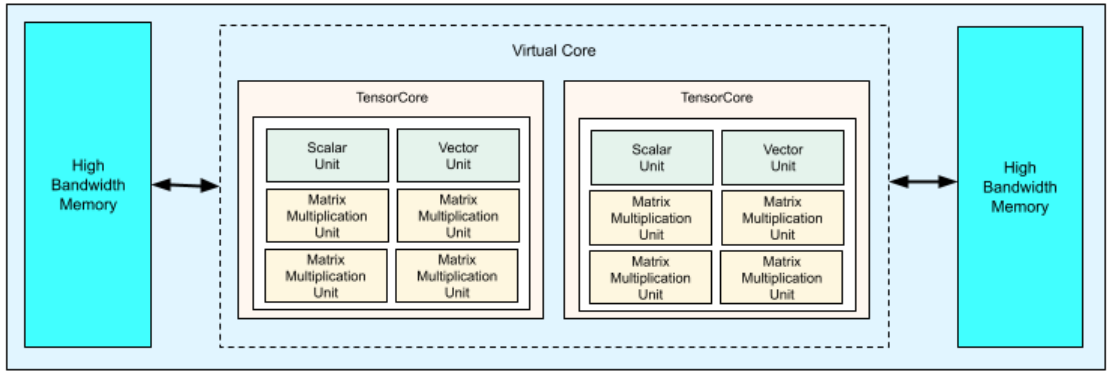
→ Various architecture



DaDianNao architecture

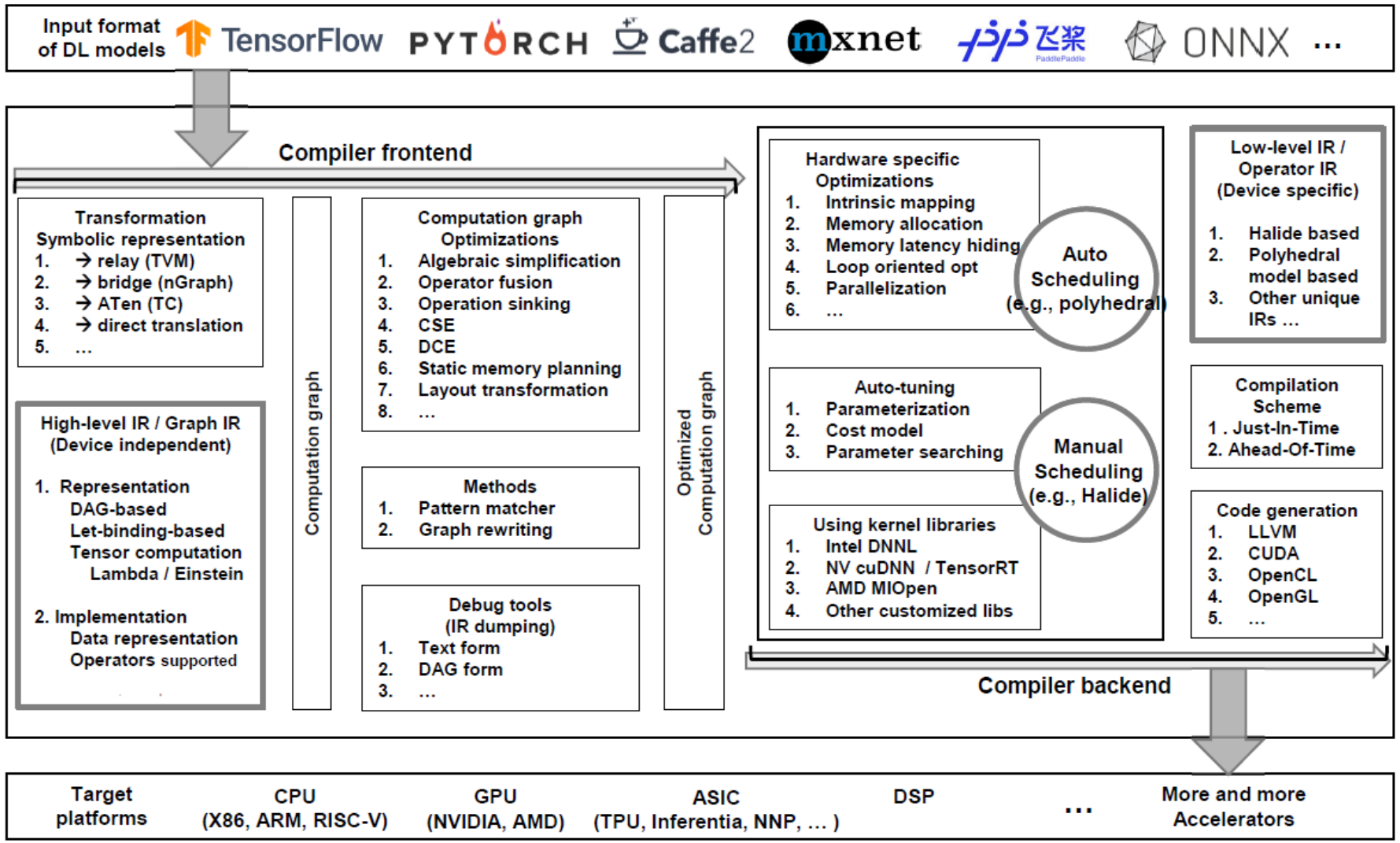


Eyeriss architecture



TPUv4 architecture

DL Compiler

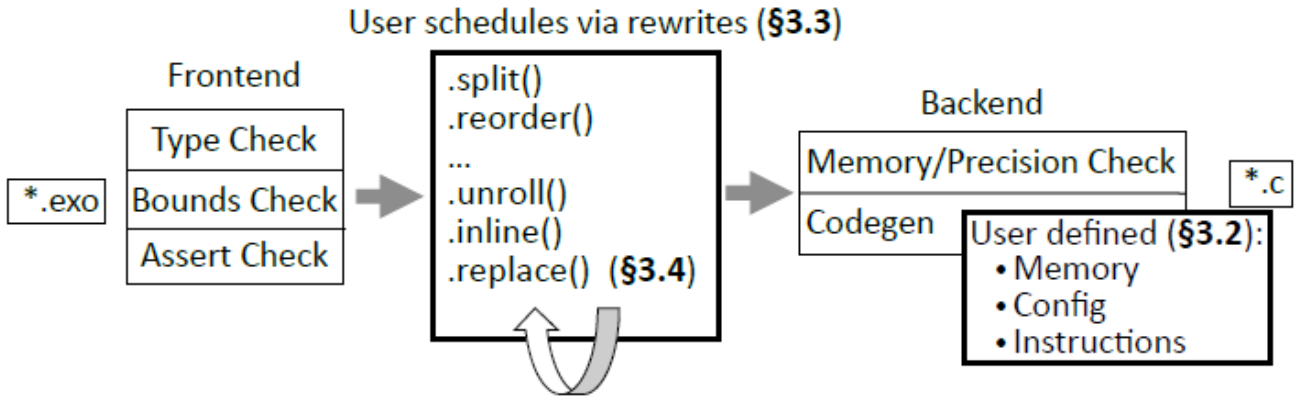

Input format of DL models
TensorFlow
PYTORCH
Caffe2
mxnet
飞桨
PaddlePaddle
ONNX
...
Compiler frontend
Transformation
Symbolic representation
1. → relay (TVM)
2. → bridge (nGraph)
3. → ATen (TC)
4. → direct translation
5. ...
High-level IR / Graph IR
(Device independent)
1. Representation
DAG-based
Let-binding-based
Tensor computation
Lambda / Einstein
2. Implementation
Data representation
Operators supported
Computation graph
Computation graph
Optimizations
1. Algebraic simplification
2. Operator fusion
3. Operation sinking
4. CSE
5. DCE
6. Static memory planning
7. Layout transformation
8. ...
Methods
1. Pattern matcher
2. Graph rewriting
Debug tools
(IR dumping)
1. Text form
2. DAG form
3. ...
Optimized Computation graph
Hardware specific Optimizations
1. Intrinsic mapping
2. Memory allocation
3. Memory latency hiding
4. Loop oriented opt
5. Parallelization
6. ...
Auto Scheduling (e.g., polyhedral)
Auto-tuning
1. Parameterization
2. Cost model
3. Parameter searching
Manual Scheduling (e.g., Halide)
Using kernel libraries
1. Intel DNNL
2. NV cuDNN / TensorRT
3. AMD MIOpen
4. Other customized libs
Low-level IR / Operator IR
(Device specific)
1. Halide based
2. Polyhedral model based
3. Other unique IRs ...
Compilation Scheme
1 . Just-In-Time
2. Ahead-Of-Time
Code generation
1. LLVM
2. CUDA
3. OpenCL
4. OpenGL
5. ...
Compiler backend
Target platforms
CPU
(X86, ARM, RISC-V)
GPU
(NVIDIA, AMD)
ASIC
(TPU, Inferentia, NNP, ...)
DSP
...
More and more Accelerators

DL Compiler

Various architecture → Various compiler backend

Exo Compiler

- Exocompilation
  - Externalize hardware specification as user-defined library
    - Special memory
    - Special instruction
    - Configuration state
  - Externalize optimization policy from compiler
    - User-scheduable language



Exo lang is python embedded language

Hardware library

custom memory

- 추상 클래스 “Memory”를 상속받은 클래스를 통해 커스텀 메모리 설계

- 사용자가 메모리 할당, 할당 해제, 접근 권한 지정 등을 설계할 수 있도록 함

```python
class GEMM_SCRATCH(Memory):
    @classmethod
    def global_(cls):
        return '#include <include/gemmini.h>\n#include "gemm_malloc.h"'

    @classmethod
    def alloc(cls, new_name, prim_type, shape, srcinfo):
        if len(shape) == 0:
            return f"{prim_type} {new_name};"

        size_str = shape[0]
        for s in shape[1:]:
            size_str = f"{s} * {size_str}"
        if not _is_const_size(shape[-1], 16):
            raise MemGenError(
                f"{srcinfo}: "
                "Cannot allocate GEMMINI Scratchpad Memory "
                "unless innermost dimension is exactly 16.  "
                f"got {shape[-1]}"
            )
        return (
            f"{prim_type} *{new_name} = "
            f"({prim_type}*) ((uint64_t)gemm_malloc ({size_str} * sizeof({prim_type})));"
        )

    @classmethod
    def can_read(cls):
        return False
```

Exo function

- @proc을 통해 알고리즘의 기본 골자를 설계
- 다음과 같은 문법을 통해 매개 변수들에 대한 설정

  <name>:<type>[<size>] @<memory>

```
@proc
def rank_k_reduce_6x16(
    K: size,
    C: f32[6, 16] @ DRAM,
    A: f32[6, K] @ DRAM,
    B: f32[K, 16] @ DRAM,
):
    for i in seq(0, 6):
        for j in seq(0, 16):
            for k in seq(0, K):
                C[i, j] += A[i, k] * B[k, j]
```

Hardware library

custom instruction/ configuration

- @instr 을 통해 명령어 설계
- 기존 컴파일러는 작성된 알고리즘을 가속기에 매핑하기 위해 추가적으로 컴파일러 백엔드를 수정해야 했지만, Exo는 라이브러리에 다음과 같이 함수로 설계 가능
- @config 를 통해 구성 설정 가능

```
@config
class ConfigLoad:
    src_stride: stride


@config
class ConfigLoad_id1:
    src_stride: stride


@config
class ConfigLoad_id2:
    src_stride: stride


@instr("gemmini_extended3_config_ld({src_stride}, 1.0f, 0, 0);\n")
def config_ld_i8(src_stride: stride):
    ConfigLoad.src_stride = src_stride
```

Loop rewrite (scheduling)

- 가능한 최적화 정책을 컴파일러에서 분할해 유저 레벨에서 관리할 수 있도록 함

- Loop nest optimization
  - Loop reordering
  - Loop fission
  - Loop fusion
- Instruction selection

Command	Transform

p.reorder(i,j)	for i:
	  for j:
	⇝
	for j:
	  for i:

p.split(i,c,io,ii)	for i<I: ⇝ for io<I/c: for ii<c:
p.unroll(i)	for i: ⇝ for 0: ...
p.inline(foo)	inline a callsite of foo in p
p.set_memory(a,MEM')	a @ MEM ⇝ a @ MEM'
p.set_precision(a,typ')	a : typ ⇝ a : typ'
p.call_eqv(foo,foo')	call foo' at a callsite of foo
p.bind_expr(a,a')	s ⇝ a' : R a' = a s[a ↦ a']

p.stage_mem(a,a',s)	s
	⇝
	a' : R[]
	for i:
	  a' = a
	s[a |↦ a']
	for i:
	  a = a'

p.bind_config(config,a)	s ⇝ config = a s[a ↦ config]

p.lift_alloc(a:R)	for i:
	  a : R
	  s
	⇝
	a : R
	for i:
	  s

p.fission_after(s1)	for i:
	  s1
	  s2
	⇝
	for i:
	  s1
	for i:
	  s2

p.reorder_stmts(s1,s2)	s1 s2 ⇝ s2 s1
p.configwrite_at(s,config,e)	s ⇝ s config = e
p.replace(s,foo)	s ⇝ foo(«inferred»)
p.add_guard(s,e)	s ⇝ if e: s else: s

p.fuse_loop(i)	for i:
	  s1
	for i:
	  s2
	⇝
	for i:
	  s1
	  s2

p.lift_if(if c: s)	for i:
	  if c: s
	⇝
	if c:
	  for i: s

p.partition_loop(i,c)	for i in lo,hi:
	⇝
	for i in lo,c:
	for i in c,hi:

p.remove_loop(i)	for i: s ⇝ s

Scheduling Example

```
# Original algorithm:
def rank_k_reduce_6x16(K: size, C: f32[6, 16] @ DRAM, A: f32[6, K] @ DRAM,
                       B: f32[K, 16] @ DRAM):
    for i in seq(0, 6):
        for j in seq(0, 16):
            for k in seq(0, K):
                C[i, j] += A[i, k] * B[k, j]
```

Scheduling Example (stage mem)

```
# First block (setting memory specification)
### C를 C_reg 버퍼에 올리고 AVX2로 매핑함
avx = rename(rank_k_reduce_6x16, "rank_k_reduce_6x16_scheduled")
avx = stage_mem(avx, 'C[_] += _', 'C[i, j]', 'C_reg')
avx = set_memory(avx, 'C_reg', AVX2)
```

```
# First block:
def rank_k_reduce_6x16_scheduled(K: size, C: f32[6, 16] @ DRAM,
                                 A: f32[6, K] @ DRAM, B: f32[K, 16] @ DRAM):
    for i in seq(0, 6):
        for j in seq(0, 16):
            for k in seq(0, K):
                C_reg: f32 @ AVX2
                C_reg = C[i, j]
                C_reg += A[i, k] * B[k, j]
                C[i, j] = C_reg
```

Scheduling Example (loop divide & reorder)

```
# Second block (loop reordering)
### j를 jo와 ji 두 루프로 변경 후 k를 기준으로 순서 변경
avx = divide_loop(avx, 'j', 8, ['jo', 'ji'], perfect=True)
avx = reorder_loops(avx, 'ji k')
avx = reorder_loops(avx, 'jo k')
avx = reorder_loops(avx, 'i k')
```

```
# Second block:
def rank_k_reduce_6x16_scheduled(K: size, C: f32[6, 16] @ DRAM,
                                 A: f32[6, K] @ DRAM, B: f32[K, 16] @ DRAM):
    for k in seq(0, K):
        for i in seq(0, 6):
            for jo in seq(0, 2):
                for ji in seq(0, 8):
                    C_reg: f32 @ AVX2
                    C_reg = C[i, 8 * jo + ji]
                    C_reg += A[i, k] * B[k, 8 * jo + ji]
                    C[i, 8 * jo + ji] = C_reg
```

Scheduling Example (lift & loop fission)

```
# Third block (loop fission)
### C_reg의 할당을 앞으로 당김
### C_reg의 초기화, 감소와 각 블록에서 write back을 진행
avx = autolift_alloc(avx, 'C_reg:_', n_lifts=4, keep_dims=True)
avx = autofission(avx, avx.find('C_reg = _ #0').after(), n_lifts=3)
avx = autofission(avx, avx.find('C_reg[_] += _ #0').after(), n_lifts=3)
avx = autofission(avx, avx.find('for i in _:_#0').after(), n_lifts=1)
avx = autofission(avx, avx.find('for i in _:_#1').after(), n_lifts=1)
avx = simplify(avx)
```

```
# Third block:
def rank_k_reduce_6x16_scheduled(K: size, C: f32[6, 16] @ DRAM,
                                 A: f32[6, K] @ DRAM, B: f32[K, 16] @ DRAM):
    C_reg: f32[1 + K, 6, 2, 8] @ AVX2
    for k in seq(0, K):
        for i in seq(0, 6):
            for jo in seq(0, 2):
                for ji in seq(0, 8):
                    C_reg[k, i, jo, ji] = C[i, ji + 8 * jo]
    for k in seq(0, K):
        for i in seq(0, 6):
            for jo in seq(0, 2):
                for ji in seq(0, 8):
                    C_reg[k, i, jo, ji] += A[i, k] * B[k, ji + 8 * jo]
    for k in seq(0, K):
        for i in seq(0, 6):
            for jo in seq(0, 2):
                for ji in seq(0, 8):
                    C[i, ji + 8 * jo] = C_reg[k, i, jo, ji]
```

Scheduling Example (stage mem & lift)

```
# Fourth block (setting memory specification)
### A를 AVX2 벡터 레지스터에 a_vec으로 설정
avx = bind_expr(avx, 'A[i, k]', 'a_vec')
avx = set_memory(avx, 'a_vec', AVX2)
avx = expand_dim(avx, 'a_vec:_', '8', 'ji')
avx = autolift_alloc(avx, 'a_vec:_')
avx = autofission(avx, avx.find('a_vec[_] = _').after())
```

```
# Fourth block:
def rank_k_reduce_6x16_scheduled(K: size, C: f32[6, 16] @ DRAM,
                                 A: f32[6, K] @ DRAM, B: f32[K, 16] @ DRAM):
    C_reg: f32[1 + K, 6, 2, 8] @ AVX2
    for k in seq(0, K):
        for i in seq(0, 6):
            for jo in seq(0, 2):
                for ji in seq(0, 8):
                    C_reg[k, i, jo, ji] = C[i, ji + 8 * jo]
    for k in seq(0, K):
        for i in seq(0, 6):
            for jo in seq(0, 2):
                a_vec: R[8] @ AVX2
                for ji in seq(0, 8):
                    a_vec[ji] = A[i, k]
                for ji in seq(0, 8):
                    C_reg[k, i, jo, ji] += a_vec[ji] * B[k, ji + 8 * jo]
    for k in seq(0, K):
        for i in seq(0, 6):
            for jo in seq(0, 2):
                for ji in seq(0, 8):
                    C[i, ji + 8 * jo] = C_reg[k, i, jo, ji]
```

Scheduling Example (stage mem & lift)

```
# Fifth block (setting memory specification)
### A를 AVX2 벡터 레지스터에 b_vec으로 설정
avx = bind_expr(avx, 'B[k, _]', 'b_vec')
avx = set_memory(avx, 'b_vec', AVX2)
avx = autolift_alloc(avx, 'b_vec:_', keep_dims=True)
avx = autofission(avx, avx.find('b_vec[_] = _').after())
```

```
# Fifth block:
def rank_k_reduce_6x16_scheduled(K: size, C: f32[6, 16] @ DRAM,
                                 A: f32[6, K] @ DRAM, B: f32[K, 16] @ DRAM):
    C_reg: f32[1 + K, 6, 2, 8] @ AVX2
    for k in seq(0, K):
        for i in seq(0, 6):
            for jo in seq(0, 2):
                for ji in seq(0, 8):
                    C_reg[k, i, jo, ji] = C[i, ji + 8 * jo]
    for k in seq(0, K):
        for i in seq(0, 6):
            for jo in seq(0, 2):
                a_vec: R[8] @ AVX2
                for ji in seq(0, 8):
                    a_vec[ji] = A[i, k]
                b_vec: R[8] @ AVX2
                for ji in seq(0, 8):
                    b_vec[ji] = B[k, ji + 8 * jo]
                for ji in seq(0, 8):
                    C_reg[k, i, jo, ji] += a_vec[ji] * b_vec[ji]
    for k in seq(0, K):
        for i in seq(0, 6):
            for jo in seq(0, 2):
                for ji in seq(0, 8):
                    C[i, ji + 8 * jo] = C_reg[k, i, jo, ji]
```

Scheduling Example (instruction select)

```
# Sixth block (replacement instruction matched)
### AVX2에 맞는 명령어로 변경
avx = replace_all(avx, avx2_set0_ps)
avx = replace_all(avx, mm256_broadcast_ss)
avx = replace_all(avx, mm256_fmadd_ps)
avx = replace_all(avx, avx2_fmadd_memu_ps)
avx = replace(avx, 'for ji in _:_ #0', mm256_loadu_ps)
avx = replace(avx, 'for ji in _:_ #0', mm256_loadu_ps)
avx = replace(avx, 'for ji in _:_ #0', mm256_storeu_ps)
```

```
# Sixth block:
def rank_k_reduce_6x16_scheduled(K: size, C: f32[6, 16] @ DRAM,
                                 A: f32[6, K] @ DRAM, B: f32[K, 16] @ DRAM):
    C_reg: f32[1 + K, 6, 2, 8] @ AVX2
    for k in seq(0, K):
        for i in seq(0, 6):
            for jo in seq(0, 2):
                mm256_loadu_ps(C_reg[k + 0, i + 0, jo + 0, 0:8],
                               C[i + 0, 8 * jo + 0:8 * jo + 8])
    for k in seq(0, K):
        for i in seq(0, 6):
            for jo in seq(0, 2):
                a_vec: R[8] @ AVX2
                mm256_broadcast_ss(a_vec, A[i + 0, k + 0:k + 1])
                b_vec: R[8] @ AVX2
                mm256_loadu_ps(b_vec[0:8], B[k + 0, 8 * jo + 0:8 * jo + 8])
                mm256_fmadd_ps(C_reg[k + 0, i + 0, jo + 0, 0:8], a_vec, b_vec)
    for k in seq(0, K):
        for i in seq(0, 6):
            for jo in seq(0, 2):
                mm256_storeu_ps(C[i + 0, 8 * jo + 0:8 * jo + 8],
                                C_reg[k + 0, i + 0, jo + 0, 0:8])
```

Scheduling Example

```
# Original algorithm:
def rank_k_reduce_6x16(K: size, C: f32[6, 16] @ DRAM, A: f32[6, K] @ DRAM,
                       B: f32[K, 16] @ DRAM):
    for i in seq(0, 6):
        for j in seq(0, 16):
            for k in seq(0, K):
                C[i, j] += A[i, k] * B[k, j]
```

```
# Sixth block:
def rank_k_reduce_6x16_scheduled(K: size, C: f32[6, 16] @ DRAM,
                                 A: f32[6, K] @ DRAM, B: f32[K, 16] @ DRAM):
    C_reg: f32[1 + K, 6, 2, 8] @ AVX2
    for k in seq(0, K):
        for i in seq(0, 6):
            for jo in seq(0, 2):
                mm256_loadu_ps(C_reg[k + 0, i + 0, jo + 0, 0:8],
                               C[i + 0, 8 * jo + 0:8 * jo + 8])
    for k in seq(0, K):
        for i in seq(0, 6):
            for jo in seq(0, 2):
                a_vec: R[8] @ AVX2
                mm256_broadcast_ss(a_vec, A[i + 0, k + 0:k + 1])
                b_vec: R[8] @ AVX2
                mm256_loadu_ps(b_vec[0:8], B[k + 0, 8 * jo + 0:8 * jo + 8])
                mm256_fmadd_ps(C_reg[k + 0, i + 0, jo + 0, 0:8], a_vec, b_vec)
    for k in seq(0, K):
        for i in seq(0, 6):
            for jo in seq(0, 2):
                mm256_storeu_ps(C[i + 0, 8 * jo + 0:8 * jo + 8],
                                C_reg[k + 0, i + 0, jo + 0, 0:8])
```

```
$ ./avx2_matmul
Time taken for original matmul: 0 seconds 649 milliseconds
Time taken for scheduled matmul: 0 seconds 291 milliseconds
```

Program Analysis

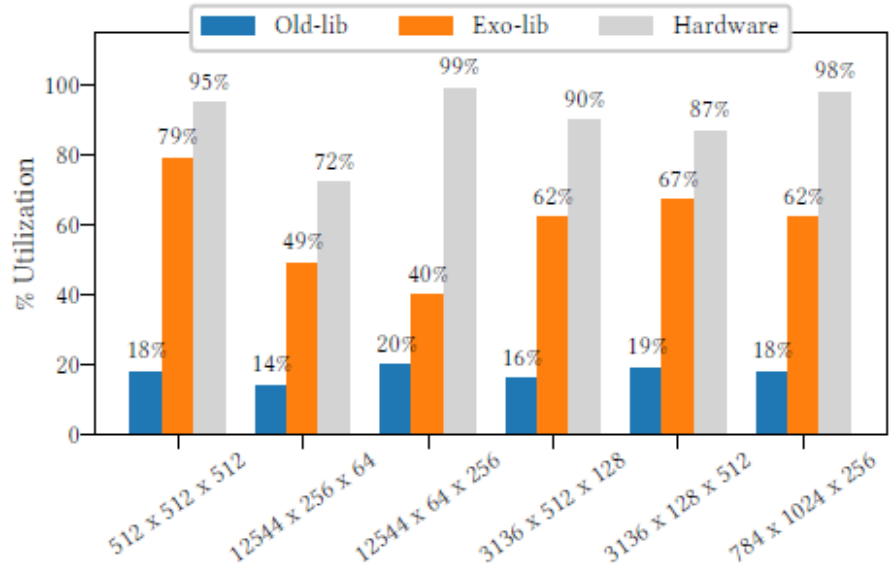
- 잘 짜여진 프로그램은 프론트엔드의 검증을 통해 재반복으로 평가함
  - 모든 정수 기반 제어 표현은 준아핀으로 제한됨
  - 버퍼에 대한 접근과 윈도우는 정적 범위에서만 진행
  - 프로시저(Exo 함수)의 호출은 호출된 프로시저의 사전 조건(Configuration)을 만족 해야함
- Exo 함수와 명령어 구성 등의 특성으로부터 영향을 추출해 이를 분석함으로 Exo 프로그램 분석을 진행하며 아래 사항들을 정의함
  - Effect-expressions and environments
  - A global symbolic data-flow analysis
  - Location sets as a symbolic abstraction of store locations
  - Effects as an abstraction of programs

τ_a : ArgType	::= `bool` | `int` | `R[`e^*`]`	
τ_s : SigType	::= $(x : \tau_a) \to \tau_s$ | `unit`	
τ : Type	::= τ_a | `R`	
note: we use $\cdot^*$ *to mean 0 or more*		
e : Expr	::= x	variables
	| $op(e^*)$	built-in operations
	| e`[`e^*`]`	array read
	| `win(`e, w^*`)`	window expression
w : WinCoord	::= e	point-access
	| e `..` e	interval-access
$op \in$	`+, -, *, /, mod, and, or, not, ==, <, <=, >, >=` $\cup$ *Literals*	
s : Stmt	::= s `;` s	sequencing
	| `if` e `then` s	guards
	| `for` x `in` e `..` e `do` s	sequential loops
	| `alloc` $x(e^*)$	array allocation
	| e`[`e^*`]= `e	array write
	| e`[`e^*`]+= `e	array reduce
	| x `=` e	global write
	| $p(e^*)$	sub-procedure call
$pdef$: Proc	::= `proc` $p : \tau_s$ `assert` e `do` s	
L : Lib	::= `globals` $(x : \tau)^*$ $pdef^*$	

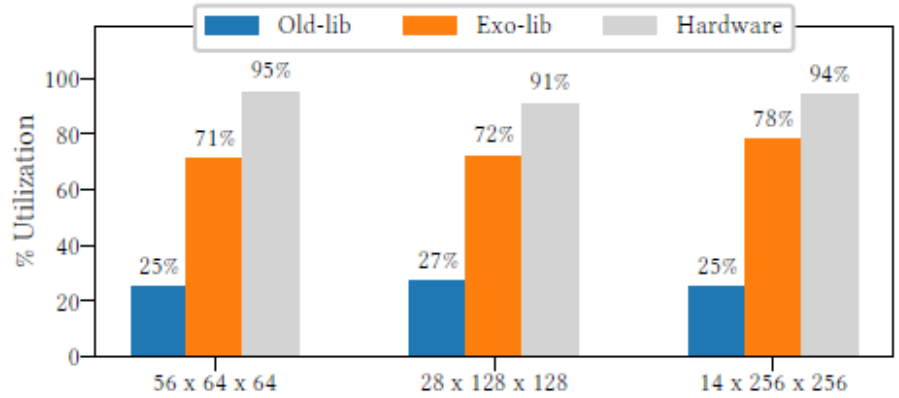
Figure 3. Abstract Syntax for Exo core language

Experiments

- 기존 Gemmini 시뮬레이터 환경의 kernel 라이브러리 대비 성능 향상
- 하드웨어적 지원을 받을 경우 추가적인 성능 향상을 기대할 수 있지만 추가적인 자원을 요구함



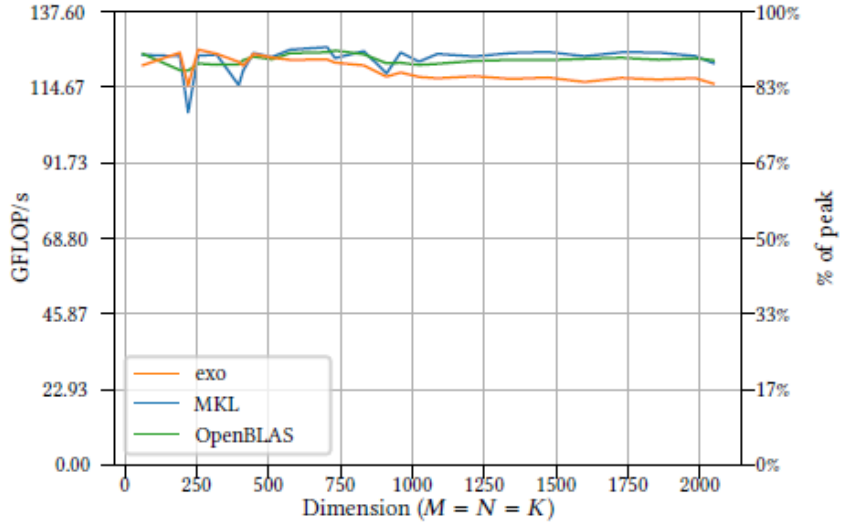
(a) MATMUL utilization (as a percentage of peak FLOPS). X axis labels are the size of matrices in *N* x *M* x *K*.



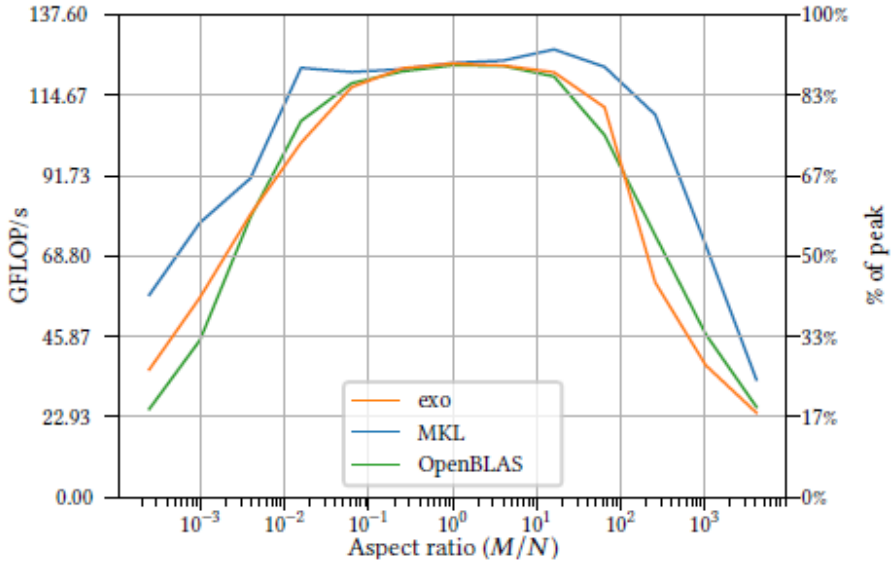
(b) CONV utilization (as a percentage of peak FLOPS). X axis labels are the shape of convolution in *output dimension* x *output channel* x *input channel*.

Experiments

- x86 시스템 벤치마크를 기준으로 기존 SOTA 커널 라이브러리들에 준하는 성능을 발휘할 수 있었음



(a) SGEMM performance on square matrices. We approximately match other systems on square matrices.



(b) SGEMM performance with fixed workload and variable output aspect ratio. $K = 512$ and $M \times N = 512^2$, with the ratio of M to N varying. We match OpenBLAS performance across aspect ratios.

Figure 5. SGEMM performance compared to state-of-the-art libraries on x86. Benchmarks were run on one core of an Intel i7-1185G7 running at 4.3GHz.

Contribution

- 하드웨어 적용을 위해 추가적인 컴파일러 변경이 아니라 유저 레벨 라이브러리를 통한 관리를 통해 빠른 적용 및 관리를 진행 가능
- 컴파일러 최적화 정책을 컴파일러로부터 분리해 유저 레벨에서 프로시저에 대한 스케줄링을 통해 최적화를 유저가 진행 가능
- 하드웨어에 대한 지원없이 추가적인 성능향상을 기대
- 프로그램 분석을 통해 하드웨어를 추상화하고 함수의 동일성과 메모리 안정성을 보장
- SOTA 커널 라이브러리의 성능에 준하는 성능 확보
  - 비용 대비 고성능 라이브러리 설계 가능

Reference

- Exocompilation for productive programming of hardware accelerators, Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. 2022.

- The deep learning compiler: A comprehensive survey, IEEE Transactions on Parallel and Distributed Systems, 2020

- Model compression and hardware acceleration for neural networks: A comprehensive survey. Proceedings of the IEEE. 2020