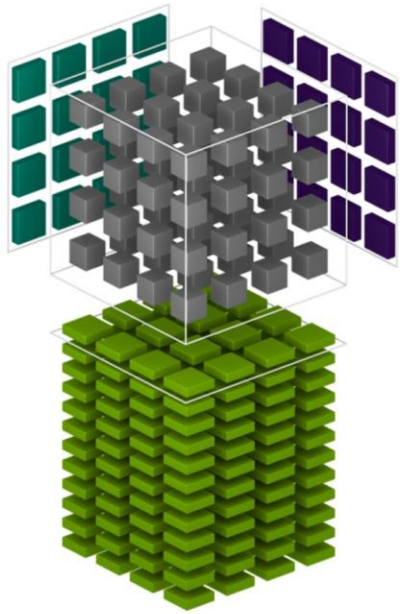
FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System

Size Zheng,Yun Liang,Shuo Wang,Renze Chen,Kaiwen Sheng

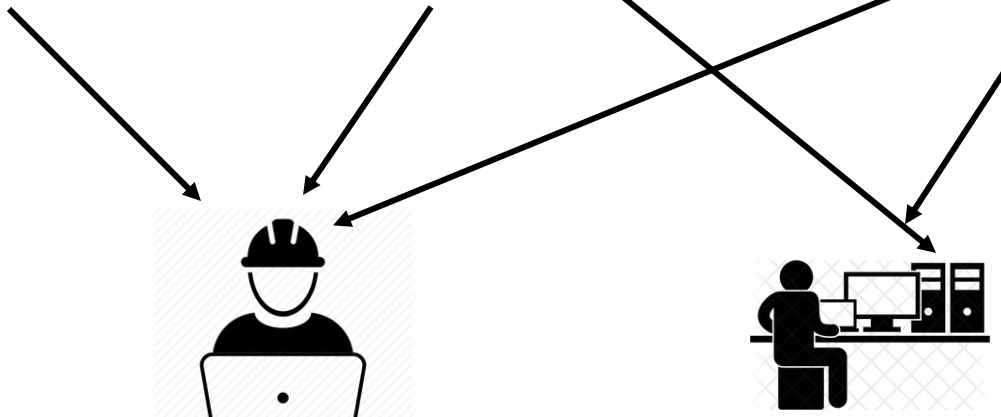
Peking University

ASPLOS'20

Complex Heterogeneous System



Tensor computation

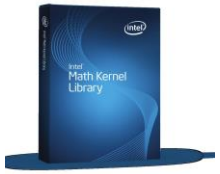


Hand-Optimized Libraries

Device

Library

Framework



cuDNN

cuBLAS

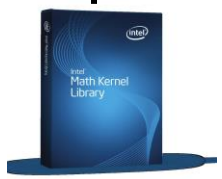
XILINX VITIS

TensorFlow

PyTorch

Caffe

Limitations of Libraries



TensorFlow

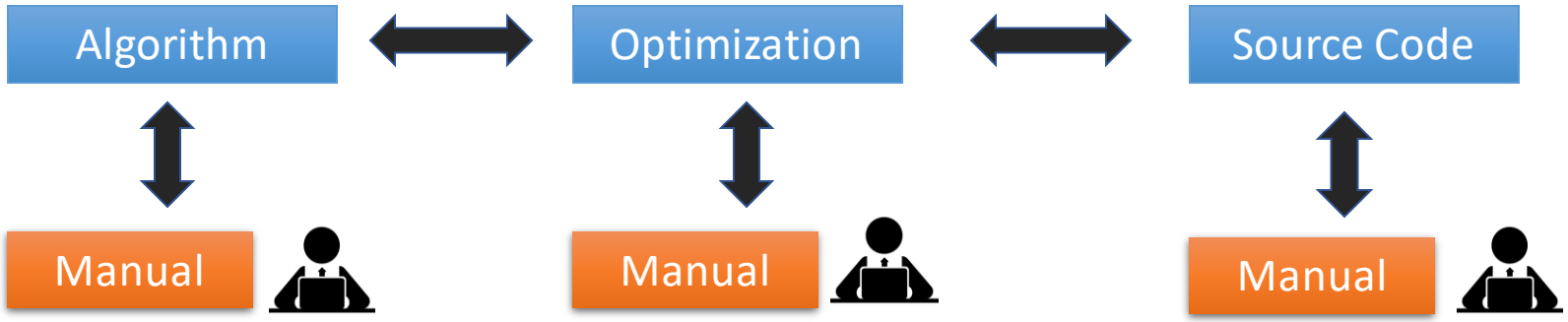
- Time cost
- Human efforts cost
- Hardly portable to other platforms

Months or years

Libaries

New applications

Limitations of Libraries



- Long time cost
- Hardware Specific
- Expertise in everything

Related Works

Halide: a language and compiler for optimizing parallelism, locality, and recomputation (PLDI'13). Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, et al.

TVM: An Automated End-to-End Optimizing Compiler for Deep Learning (OSDI'18). Tianqi Chen, Thierry Moreau, Ziheng Jiang, et al.

Related Works(Halide)

- Separate Compute and Schedule
- Focus on image-processing
- Autoscheduler mainly for CPU
- Generate low-level code automatically

Related Works(TVM)

- Separate Compute and Schedule
- Focus on machine-learning
- AutoTVM requires hand-optimized template for auto-tuning
- Generate low-level code automatically

Related Works(TVM)

```python
def schedule_direct_cuda(cfg, s, conv):
    """schedule optimized for batch size = 1"""

    ##### space definition begin #####
    n, f, y, x = s[conv].op.axis
    rc, ry, rx = s[conv].op.reduce_axis
    cfg.define_split("tile_f", f, num_outputs=4)
    cfg.define_split("tile_y", y, num_outputs=4)
    cfg.define_split("tile_x", x, num_outputs=4)
    cfg.define_split("tile_rc", rc, num_outputs=2)
    cfg.define_split("tile_ry", ry, num_outputs=2)
    cfg.define_split("tile_rx", rx, num_outputs=2)
    cfg.define_knob("auto_unroll_max_step", [0, 512, 1500])

    target = tvm.target.Target.current()
    if target.target_name in ['nvptx', 'rocm']:
        cfg.define_knob("unroll_explicit", [1])
    else:
        cfg.define_knob("unroll_explicit", [0, 1])

    # fallback support
    if cfg.is_fallback:
        ref_log = autotvm.tophub.load_reference_log(
            target.target_name, target.model, 'conv2d', 'direct')
        cfg.fallback_with_reference_log(ref_log)
    ##### space definition end #####

    pad_data, kernel = s[conv].op.input_tensors

    s[pad_data].compute_inline()
    if isinstance(kernel.op, tvm.tensor.ComputeOp) and 'dilate' in kernel.op.tag:
        s[kernel].compute_inline()

    if conv.op in s.outputs:
        output = conv
        OL = s.cache_write(conv, 'local')
    else:
        output = s.outputs[0].output(0)
        s[conv].set_scope('local')
        OL = conv

    # create cache stage
    AA = s.cache_read(pad_data, 'shared', [OL])
    WW = s.cache_read(kernel, 'shared', [OL])

    # tile and bind spatial axes
    n, f, y, x = s[output].op.axis
    kernel_scope, n = s[output].split(n, nparts=1)

    bf, vf, tf, fi = cfg["tile_f"].apply(s, output, f)
    by, vy, ty, yi = cfg["tile_y"].apply(s, output, y)
    bx, vx, tx, xi = cfg["tile_x"].apply(s, output, x)
```

TVM template example

	height	width	Cin	Cout	stride	kernel	padding	dilation	#config
0	7	7	512	512	(1, 1)	(3, 3)	(1, 1)	(1, 1)	844800
1	14	14	256	512	(2, 2)	(3, 3)	(1, 1)	(1, 1)	760320
2	14	14	256	256	(1, 1)	(3, 3)	(1, 1)	(1, 1)	9123840
3	28	28	128	256	(2, 2)	(3, 3)	(1, 1)	(1, 1)	8110080
4	28	28	128	128	(1, 1)	(3, 3)	(1, 1)	(1, 1)	36864000
5	56	56	64	128	(2, 2)	(3, 3)	(1, 1)	(1, 1)	32256000
6	56	56	64	64	(1, 1)	(3, 3)	(1, 1)	(1, 1)	90316800
7	224	224	3	64	(2, 2)	(7, 7)	(3, 3)	(1, 1)	79027200
8	56	56	64	64	(1, 1)	(1, 1)	(0, 0)	(1, 1)	22579200
9	56	56	64	128	(2, 2)	(1, 1)	(0, 0)	(1, 1)	8064000
10	28	28	128	256	(2, 2)	(1, 1)	(0, 0)	(1, 1)	2027520
11	14	14	256	512	(2, 2)	(1, 1)	(0, 0)	(1, 1)	190080

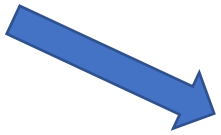
	tile_y_num_ou	x_num_ou	x_num_ou	y_num_ou	f_num_ou	max_step	xplicit_num	rc_num_ou	e_y_entitie	e_rx_entitie	e_x_entitie	e_ry_entitie	e_f_entitie	l_max_ste	explicit_e	e_rc_entities
0	4	2	4	2	220	3	2	10	[[7, 1, 1, 1	[[3, 1], [1,	[[7, 1, 1, 1	[[3, 1], [1,	[[512, 1, 1	[0, 512, 15	[0, 1]	[[512, 1], [256, 2], [12
1	4	2	4	2	220	3	2	9	[[7, 1, 1, 1	[[3, 1], [1,	[[7, 1, 1, 1	[[3, 1], [1,	[[512, 1, 1	[0, 512, 15	[0, 1]	[[256, 1], [128, 2], [64
2	16	2	16	2	165	3	2	9	[[14, 1, 1,	[[3, 1], [1,	[[14, 1, 1,	[[3, 1], [1,	[[256, 1, 1	[0, 512, 15	[0, 1]	[[256, 1], [128, 2], [64
3	16	2	16	2	165	3	2	8	[[14, 1, 1,	[[3, 1], [1,	[[14, 1, 1,	[[3, 1], [1,	[[256, 1, 1	[0, 512, 15	[0, 1]	[[128, 1], [64, 2], [32,
4	40	2	40	2	120	3	2	8	[[28, 1, 1,	[[3, 1], [1,	[[28, 1, 1,	[[3, 1], [1,	[[128, 1, 1	[0, 512, 15	[0, 1]	[[128, 1], [64, 2], [32,
5	40	2	40	2	120	3	2	7	[[28, 1, 1,	[[3, 1], [1,	[[28, 1, 1,	[[3, 1], [1,	[[128, 1, 1	[0, 512, 15	[0, 1]	[[64, 1], [32, 2], [16, 4
6	80	2	80	2	84	3	2	7	[[56, 1, 1,	[[3, 1], [1,	[[56, 1, 1,	[[3, 1], [1,	[[64, 1, 1,	[0, 512, 15	[0, 1]	[[64, 1], [32, 2], [16, 4
7	140	2	140	2	84	3	2	2	[[112, 1, 1	[[7, 1], [1,	[[112, 1, 1	[[7, 1], [1,	[[64, 1, 1,	[0, 512, 15	[0, 1]	[[3, 1], [1, 3]]
8	80	1	80	1	84	3	2	7	[[56, 1, 1,	[[1, 1]]	[[56, 1, 1,	[[1, 1]]	[[64, 1, 1,	[0, 512, 15	[0, 1]	[[64, 1], [32, 2], [16, 4
9	40	1	40	1	120	3	2	7	[[28, 1, 1,	[[1, 1]]	[[28, 1, 1,	[[1, 1]]	[[128, 1, 1	[0, 512, 15	[0, 1]	[[64, 1], [32, 2], [16, 4
10	16	1	16	1	165	3	2	8	[[14, 1, 1,	[[1, 1]]	[[14, 1, 1,	[[1, 1]]	[[256, 1, 1	[0, 512, 15	[0, 1]	[[128, 1], [64, 2], [32,
11	4	1	4	1	220	3	2	9	[[7, 1, 1, 1	[[1, 1]]	[[7, 1, 1, 1	[[1, 1]]	[[512, 1, 1	[0, 512, 15	[0, 1]	[[256, 1], [128, 2], [64

available tuning options in tvm(resnet-18)

Code-Generation with Scheduling

compute description

```
def vector_add ( A, B ):
    C = compute ( ( 16, ),
    lambda i : A [ i ] + B [ i ] )
return C
```

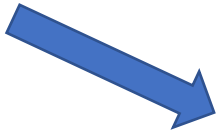


naive source code

```
for ( int i=0; i < 16; i=i+1)
{
    C [ i ] = A [ i ] + B [ i ];
}
```

scheduling

```
C = vector_add ( A, B )
s = create_schedule ( C.op )
i = s [C].op.axis [0]
outer, inner =
    s [C].split ( i, factor=4 )
s [ C ].unroll ( inner )
```



optimized code

```
for (int outer=0; outer < 4; outer=outer+1)
{
    C [outer*4+0]= A [outer*4+0]+B [outer*4+0];
    C [outer*4+1]= A [outer*4+1]+B [outer*4+1];
    C [outer*4+2]= A [outer*4+2]+B [outer*4+2];
    C [outer*4+3]= A [outer*4+3]+B [outer*4+3];
}
```

Code-Generation with Scheduling

Compute Description

- High-level
- Algorithm
- Mathematic expressions

Scheduling

- Primitives
- Hardware-specific
- Parameters for optimization

Table 2. Different schedule primitives for different target platforms and their parameters.

Target	Name	Description	Parameter
All	split	divide a loop into several sub-loops	loop to split and split factors
	fuse	merge several loops into a hyper-loop	adjacent loop to fuse
	reorder	change execution orders of loops	loops to reorder and new order
	unroll	unroll a loop by given depth	which loop to unroll and unroll depth
	vectorize	apply vector operation to a loop	which loop to vectorize
	inline	inline a function	which node to inline
	compute at	put producer in the body of consumer	which node and how deep to compute at
CPU	parallel	use multithreading	which loop to parallel
GPU	cache	use shared memory to store inputs/results	which tensor to cache and how much data to cache
	bind	assign a loop to parallel blocks/threads	which loop to bind to block/thread
FPGA	buffer	how much input to buffer at a time	rows and columns of inputs to buffer
	pipeline	pipeline of data read/write and computation.	number of stages in pipeline and number of pipelines
	partition	memeory partition to increase available bandwidth	partion number

Recap : Locality of reference

Princicle of Locality

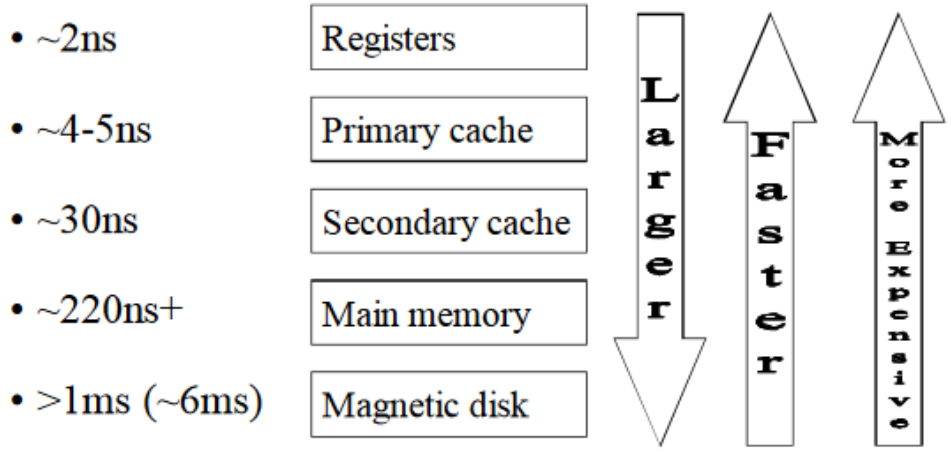
- programs tend to use data and instuctions with address near or equal to thosr they have used recently

Temporal locality

-Recently referenced items are likely to be referenced again in the near future

Spatial locality

-Items with nearby addresses tend to be referenced clost together in time



Loop Transforms(without dependency)

```cpp
# include<vector>
int main(void)
{
    const int size=10;
    std::vector<std::vector<int>>  vec1(size, std::vector<int>(size,3));
    std::vector<std::vector<int>>  vec2(size, std::vector<int>(size,3));
    // no data dependency !
    for (int i=0;i<5;i++)
    {
        vec1[i][0]=vec2[i][0]+vec1[i][0];
        vec1[i][1]=vec2[i][1]+vec1[i][1];
        vec1[i][2]=vec2[i][2]+vec1[i][2];
        vec1[i][3]=vec2[i][3]+vec1[i][3];
        vec1[i][4]=vec2[i][4]+vec1[i][4];
        /*
        vec1[i][1]=vec2[i][1]+vec1[i][1];
        vec1[i][0]=vec2[i][0]+vec1[i][0];
        vec1[i][3]=vec2[i][3]+vec1[i][3];
        vec1[i][2]=vec2[i][2]+vec1[i][2];
        vec1[i][4]=vec2[i][4]+vec1[i][4];
        */
    }
    return 0;
}
```

```cpp
# include<vector>
int main(void)
{
    const int size=10;
    std::vector<std::vector<int>>  vec1(size, std::vector<int>(size,3));
    std::vector<std::vector<int>>  vec2(size, std::vector<int>(size,3));
    // no data dependency !
    for (int i=0;i<5;i++)
    {
        for (int j=0;j<5;j++)
        {
            vec1[i][j]=vec2[i][j]+vec1[i][j];
        }
    }
    return 0;
}
```

Loop Transforms(with dependency)

```cpp
# include<vector>
int main(void)
{
    const int size=10;
    std::vector<std::vector<int>>  vec1(size, std::vector<int>(size,3));
    std::vector<std::vector<int>>  vec2(size, std::vector<int>(size,3));
    // data dependency !
    for (int i=0;i<5;i++)
    {
        vec1[i][1]=vec2[i][0]+vec1[i][0];
        vec1[i][2]=vec2[i][1]+vec1[i][1];
        vec1[i][3]=vec2[i][2]+vec1[i][2];
        vec1[i][4]=vec2[i][3]+vec1[i][3];
        vec1[i][5]=vec2[i][4]+vec1[i][4];
        /*
        vec1[i][2]=vec2[i][1]+vec1[i][1];
        vec1[i][1]=vec2[i][0]+vec1[i][0];
        vec1[i][4]=vec2[i][3]+vec1[i][3];
        vec1[i][3]=vec2[i][2]+vec1[i][2];
        vec1[i][5]=vec2[i][4]+vec1[i][4];
        */
    }
    return 0;
}
```

```cpp
# include<vector>
int main(void)
{
    const int size=10;
    std::vector<std::vector<int>>  vec1(size, std::vector<int>(size,3));
    std::vector<std::vector<int>>  vec2(size, std::vector<int>(size,3));
    // data dependency !
    for (int i=0;i<5;i++)
    {
        for (int j=0;j<5;j++)
        {
            vec1[i][j+1]=vec2[i][j]+vec1[i][j];
        }
    }
    return 0;
}
```

Loop Interchange

```cpp
# include<vector>
int main(void)
{
    const int size=10;
    std::vector<std::vector<int>>  vec1(size, std::vector<int>(size,3));
    std::vector<std::vector<int>>  vec2(size, std::vector<int>(size,3));
    // Loop Interchange !
    for (int j=0;j<5;j++)
    {
        vec1[0][j+1]=vec2[0][j]+vec1[0][j];
        vec1[1][j+1]=vec2[1][j]+vec1[1][j];
        vec1[2][j+1]=vec2[2][j]+vec1[2][j];
        vec1[3][j+1]=vec2[3][j]+vec1[3][j];
        vec1[4][j+1]=vec2[4][j]+vec1[4][j];
        /*
        vec1[2][j+1]=vec2[2][j]+vec1[2][j];
        vec1[0][j+1]=vec2[0][j]+vec1[0][j];
        vec1[1][j+1]=vec2[1][j]+vec1[1][j];
        vec1[4][j+1]=vec2[4][j]+vec1[4][j];
        vec1[3][j+1]=vec2[3][j]+vec1[3][j];
        */
    }
    return 0;
}
```

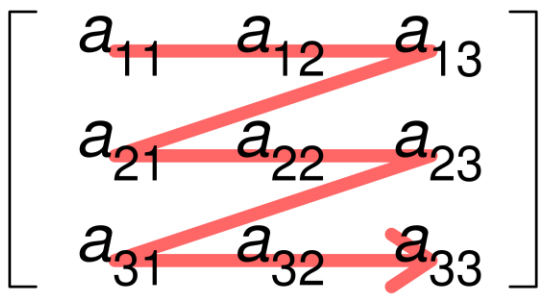
```cpp
# include<vector>
int main(void)
{
    const int size=10;
    std::vector<std::vector<int>>  vec1(size, std::vector<int>(size,3));
    std::vector<std::vector<int>>  vec2(size, std::vector<int>(size,3));
    // Loop Interchange !
    for (int j=0;j<5;j++)
    {
        for (int i=0;i<5;i++)
        {
            vec1[i][j+1]=vec2[i][j]+vec1[i][j];
        }
    }
    return 0;
}
```

Improve data locality & Increase parallelism

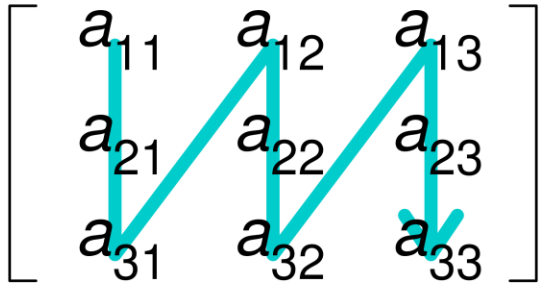
Loop Interchange

```cpp
# include<vector>
int main(void)
{
    const int size=10;
    std::vector<std::vector<int>>  vec1(size, std::vecto
    std::vector<std::vector<int>>  vec2(size, std::vecto
    // Loop Interchange !
    for (int j=0;j<5;j++)
    {
        vec1[0][j+1]=vec2[0][j]+vec1[0][j];
        vec1[1][j+1]=vec2[1][j]+vec1[1][j];
        vec1[2][j+1]=vec2[2][j]+vec1[2][j];
        vec1[3][j+1]=vec2[3][j]+vec1[3][j];
        vec1[4][j+1]=vec2[4][j]+vec1[4][j];
        /*
        vec1[2][j+1]=vec2[2][j]+vec1[2][j];
        vec1[0][j+1]=vec2[0][j]+vec1[0][j];
        vec1[1][j+1]=vec2[1][j]+vec1[1][j];
        vec1[4][j+1]=vec2[4][j]+vec1[4][j];
        vec1[3][j+1]=vec2[3][j]+vec1[3][j];
        */
    }
    return 0;
}
```

Row-major order



Column-major order

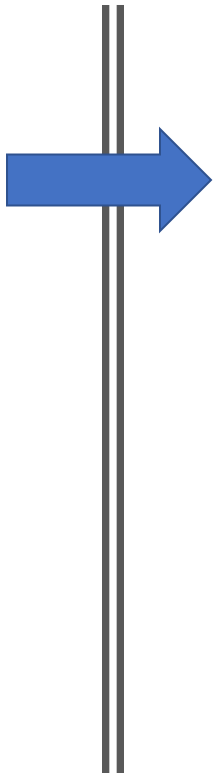



```cpp
ector>
id)

nt size=10;
ctor<std::vector<int>>  vec1(size, std::vector<int>(size,3));
ctor<std::vector<int>>  vec2(size, std::vector<int>(size,3));
 Interchange !
t j=0;j<5;j++)

(int i=0;i<5;i++)

vec1[i][j+1]=vec2[i][j]+vec1[i][j];


0;
```

Improve data locality & Increase parallelism

Loop Distribution(split)

```cpp
# include<vector>
int main(void)
{
    const int size=1000000000;
    std::vector<int>  vec1(size, 3);
    std::vector<int>  vec2(size, 3);
    std::vector<int>  vec3(size, 3);
    std::vector<int>  vec4(size, 3);

    // Loop Distribution !
    for (int i=0;i<5;i++)
    {
       vec1[i] = vec2[i] + vec3[i];
       vec4[i] = vec1[i] + 2;
       vec2[i+1]=vec1[i]*3;
    }

    return 0;
}
```

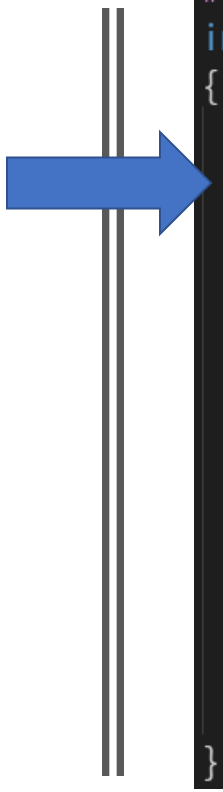


```cpp
# include<vector>
int main(void)
{
    const int size=1000;
    std::vector<int>  vec1(size, 3);
    std::vector<int>  vec2(size, 3);
    std::vector<int>  vec3(size, 3);
    std::vector<int>  vec4(size, 3);

    // Loop Distribution !
    for (int i=0;i<5;i++)
    {
       vec1[i] = vec2[i] + vec3[i];
       vec2[i+1]=vec1[i]*3;
    }
    for (int i=0;i<5;i++)
    {
        vec4[i] = vec1[i] + 2;
    }
    return 0;
}
```

Better locality & Isolate parallelizable loops

Loop Fusion

```cpp
# include<vector>
int main(void)
{
    const int size=10;
    std::vector<int>  vec1(size, 3);
    std::vector<int>  vec2(size, 3);
    std::vector<int>  vec3(size, 3);

    // Loop Distribution !
    for (int i=0;i<5;i++)
    {
       vec1[i] = vec2[i] + vec3[i];
    }
    for (int i=0;i<5;i++)
    {
        vec2[i]  = vec1[i]*3;
    }
    return 0;
}
```

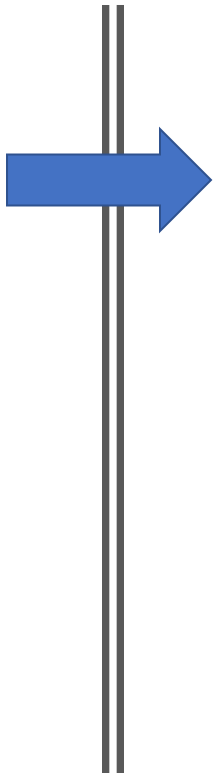


```cpp
# include<vector>
int main(void)
{
    const int size=10;
    std::vector<int>  vec1(size, 3);
    std::vector<int>  vec2(size, 3);
    std::vector<int>  vec3(size, 3);

    // Loop Distribution !
    for (int i=0;i<5;i++)
    {
       vec1[i] = vec2[i] + vec3[i];
       vec2[i]  = vec1[i]*3;
    }

    return 0;
}
```

Reduce loop overheads & Better locality & Increase the granularity of parallelism

Loop Unroll

```cpp
# include<vector>
int main(void)
{
    const int size=100;
    std::vector<int>  vec1(size, 3);
    std::vector<int>  vec2(size, 3);
    std::vector<int>  vec3(size, 3);

    // Loop Unroll !
    for (int i=0;i<100;i++)
    {
       vec1[i] = vec2[i] + vec3[i];
    }

    return 0;
}
```

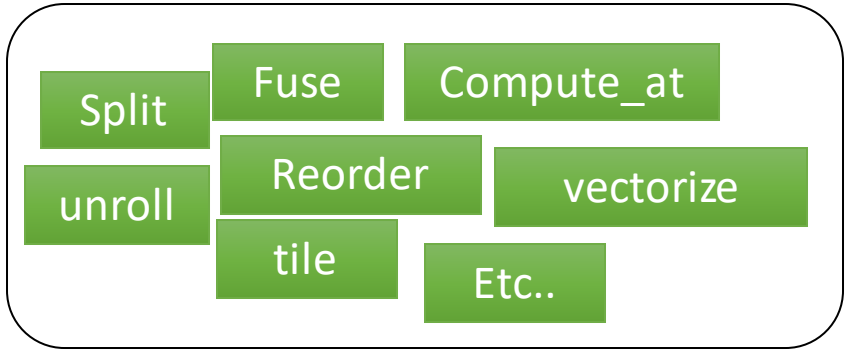


```cpp
# include<vector>
int main(void)
{
    const int size=100;
    std::vector<int>  vec1(size, 3);
    std::vector<int>  vec2(size, 3);
    std::vector<int>  vec3(size, 3);

    // Loop Unroll !
    for (int i=0;i<100;i+=4)
    {
       vec1[i] = vec2[i] + vec3[i];
       vec1[i+1] = vec2[i+1] + vec3[i+1];
       vec1[i+2] = vec2[i+2] + vec3[i+2];
       vec1[i+3] = vec2[i+3] + vec3[i+3];
    }

    return 0;
}
```

Reduce the overheads of induction variable update and conditional branches

Limitations of Writing Schedules

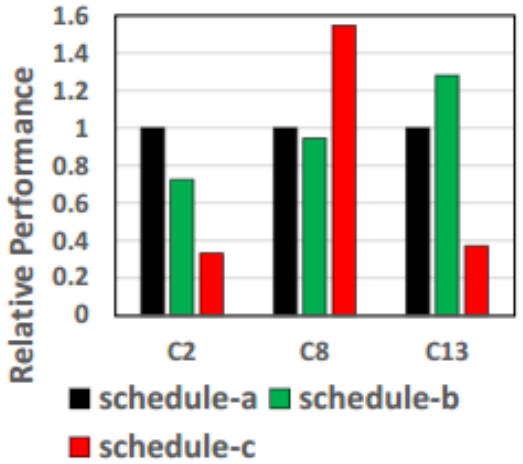
- **Many primitives to choose**



- **Complex combinations & huge parameter space**

Search space > 10^11

Limitations of Writing Schedules



- Different input scale
- Different primitive combination

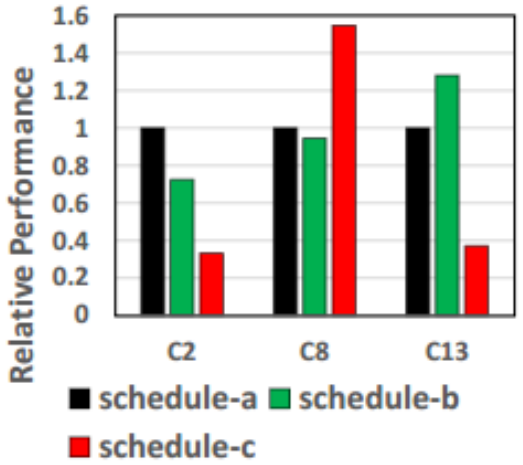
schedule

- a:split
- b:bind
- c:fuse

compute description (conv2d)

- C2 : input 64 output 192 H/W 112 kernel 3 stride 1 (schedule a => best)
- C8 : input 256 output 512 H/W 28 kernel 3 stride 1 (schedule c => best)
- C13 : input 1024 output 1024 H/W 14 kernel 3 stride 1 (schedule b => best)

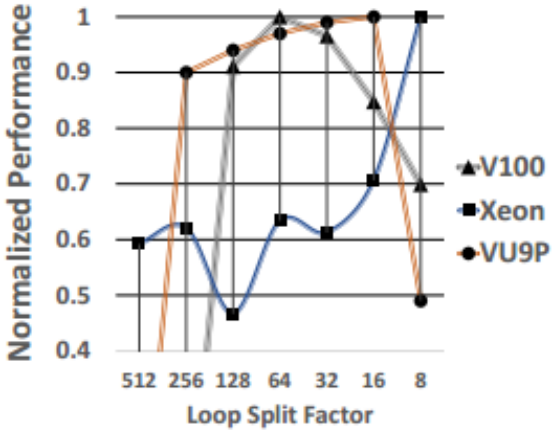
Limitations of Writing Schedules



- Different input scale
- Different primitive combination(a:split,b:bind,c:fuse)

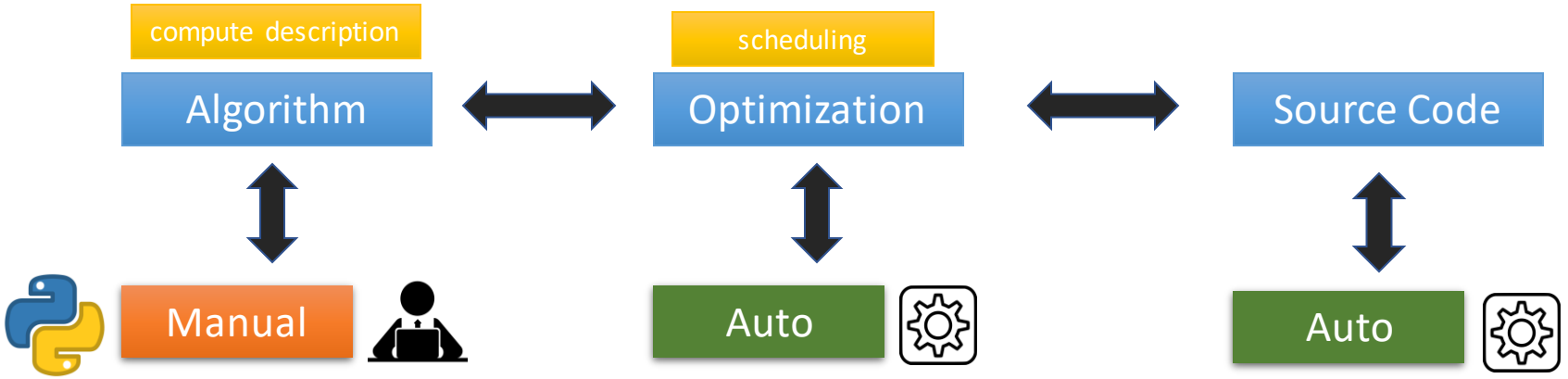
- V100 => 64 is best
- Xeon => 8 is best
- VU9P=> 16 is best

- Different parameters
- Different hardware



the inner loop for 2D convolution

FlexTensor



focus on algorithm

hide hardware details from users

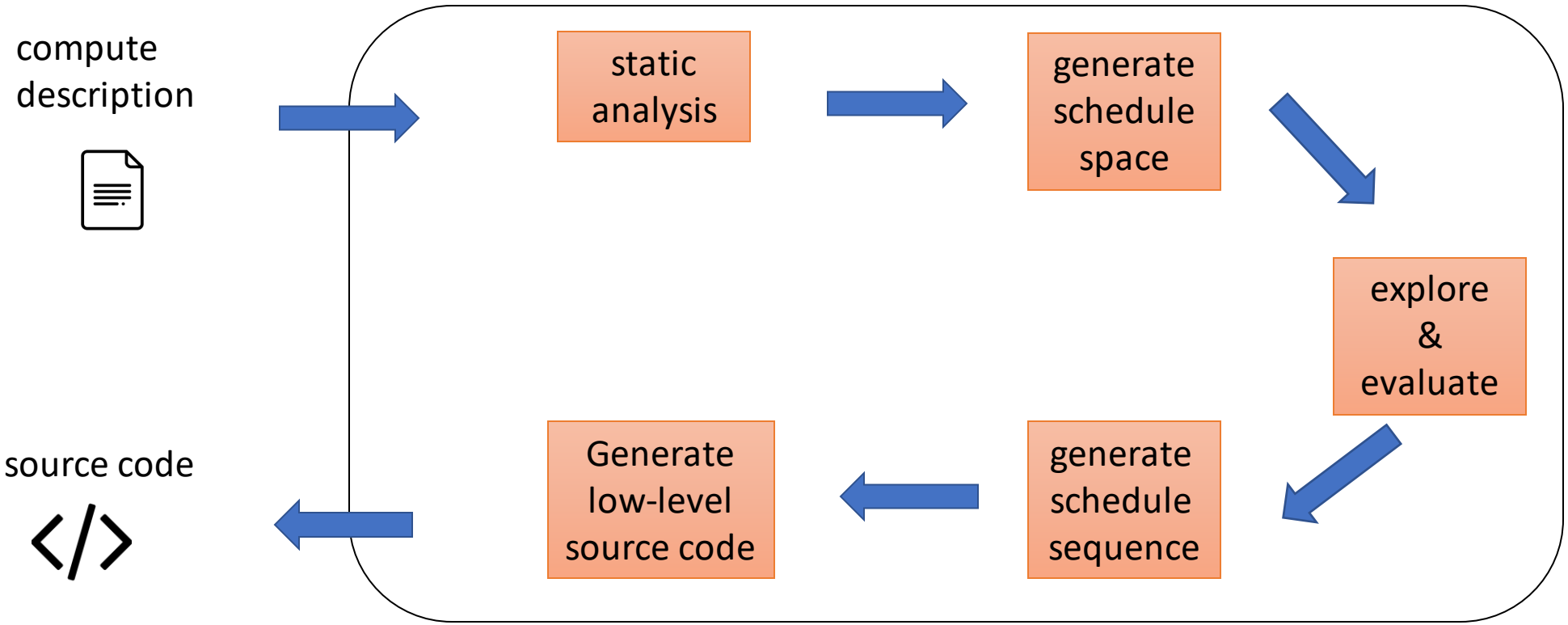
only expertise in algorithm

FlexTensor

Key Idea: replace hand-written schedules with **automatic schedule exploration and optimization**

Motivation	Ideal	FlexTensor
Optimization	Auto	**Automatic** schedule exploration
Portable	Yes	Support CPU, GPU, FPGA
Performance	High	Speedup: **1.83x** on GPU; **1.72x** on CPU; **1.5x** on FPGA
Programming	High level	Only compute description in Python
Development time	Short	10 min – 1 hour

FlexTensor Workflow



Static Analysis

```
def vector_add ( A, B ):
    C = compute ( ( 16, ),
    lambda i : A [ i ] + B [ i ] )
return C
```

compute description

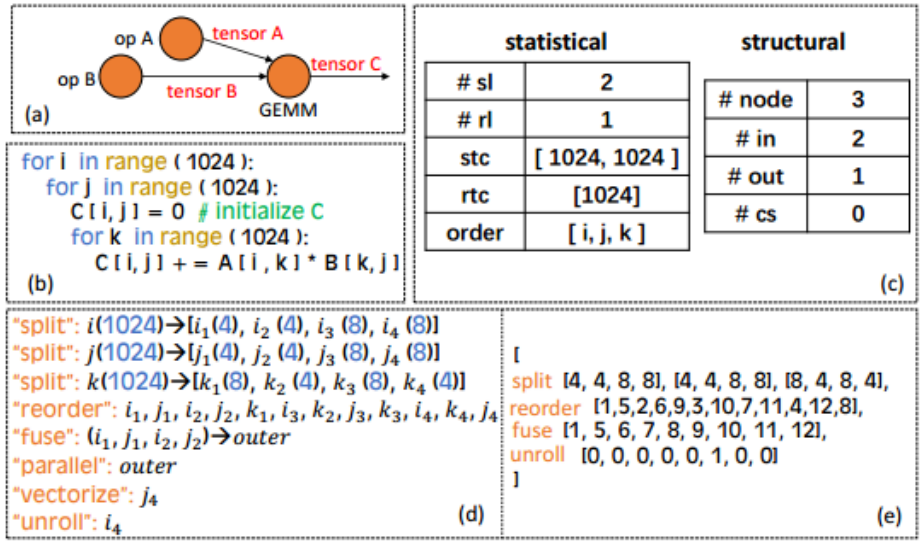
statistical information

- loop trip counts
- number of loops
- loop order
- loop type
  - Spatial loops(without data dependency)
  - Reduce loops(data dependency)

structural information

- computation graph structure
- producer consumer relationship

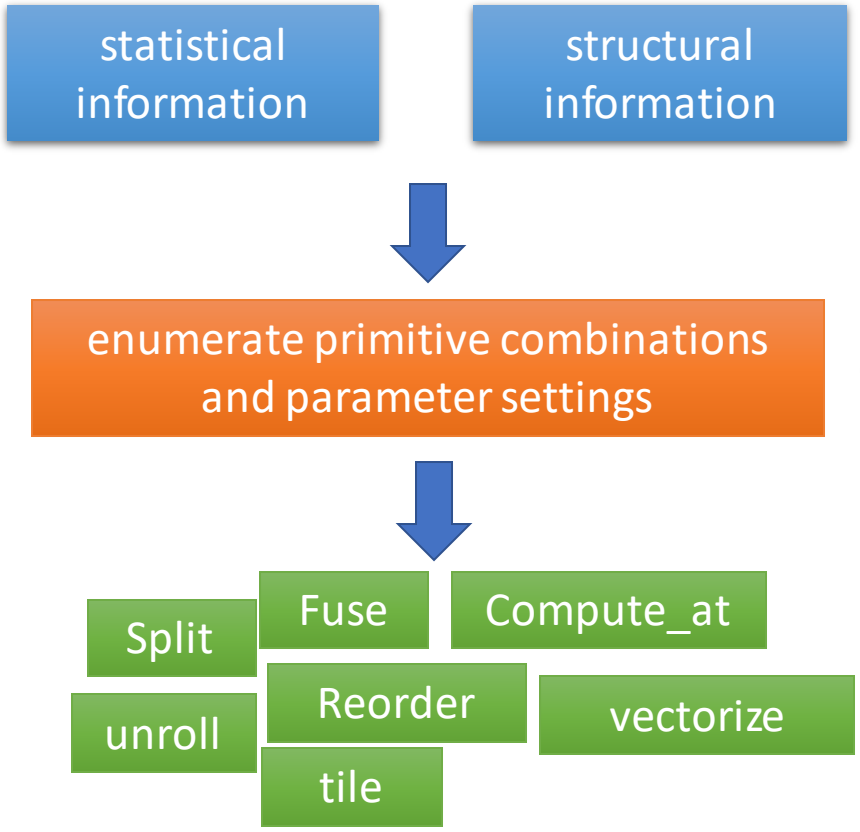
Static Analysis



An example of GEMM

- (a) GEMM mini-graph
- (b) GEMM high-level code
- (c) Statistical & structural info. from code (b)

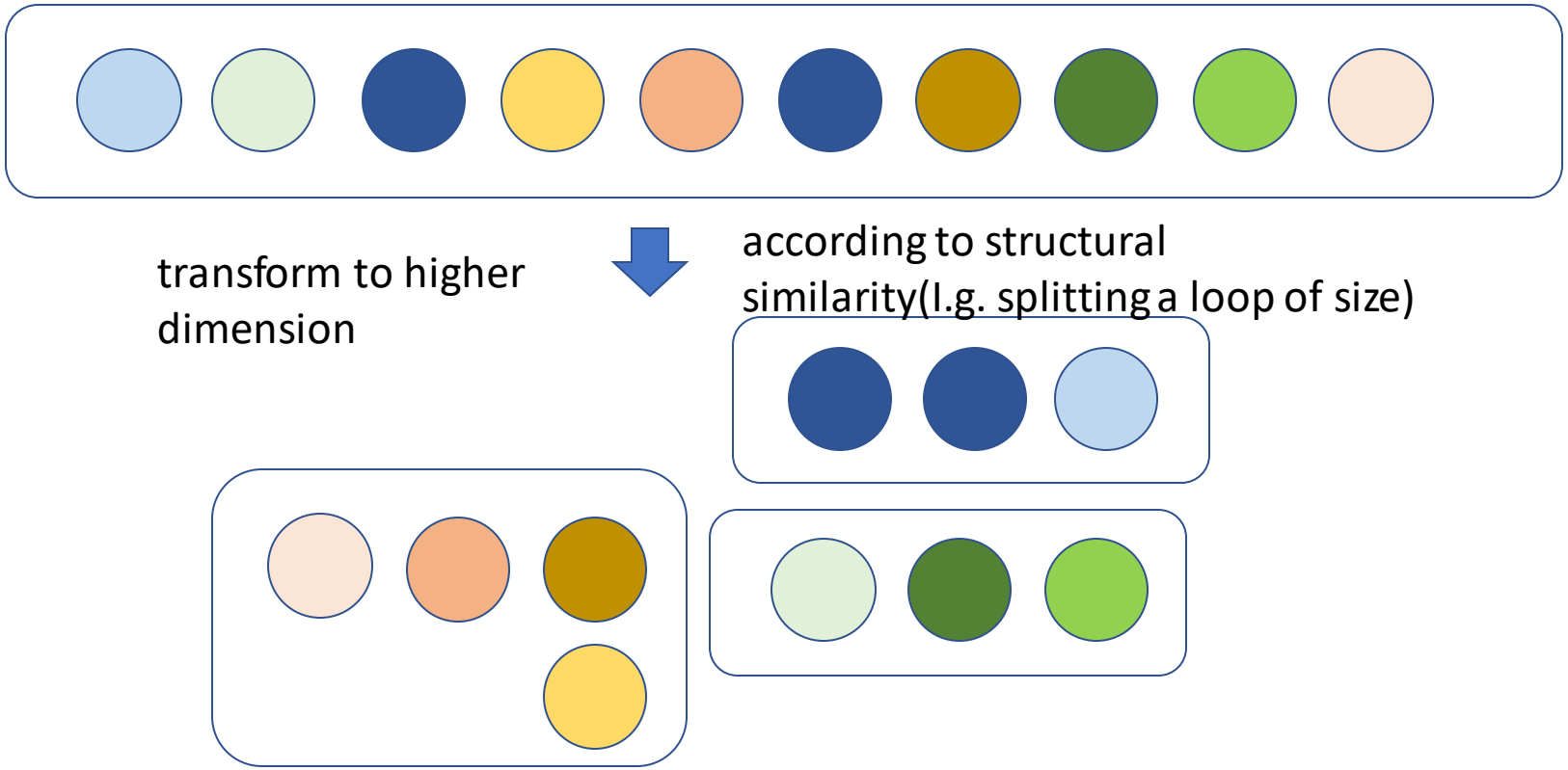
Schedule Space Generation



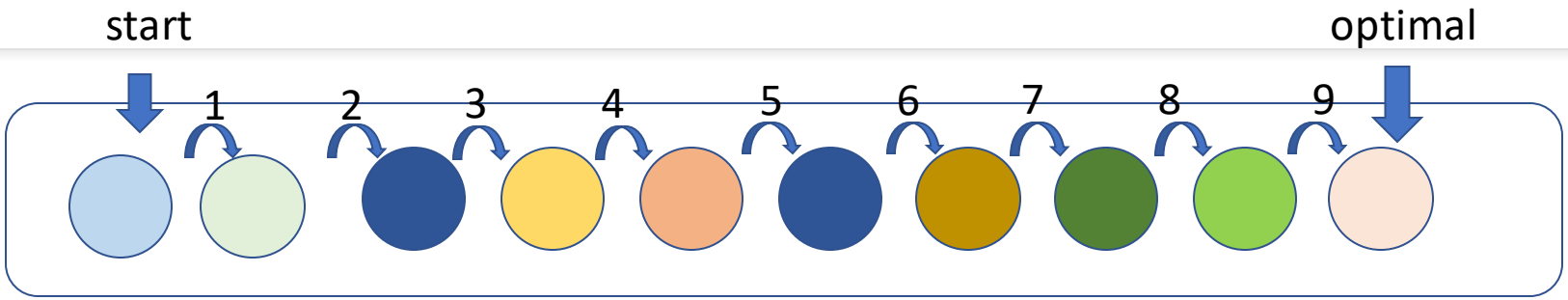
Principles

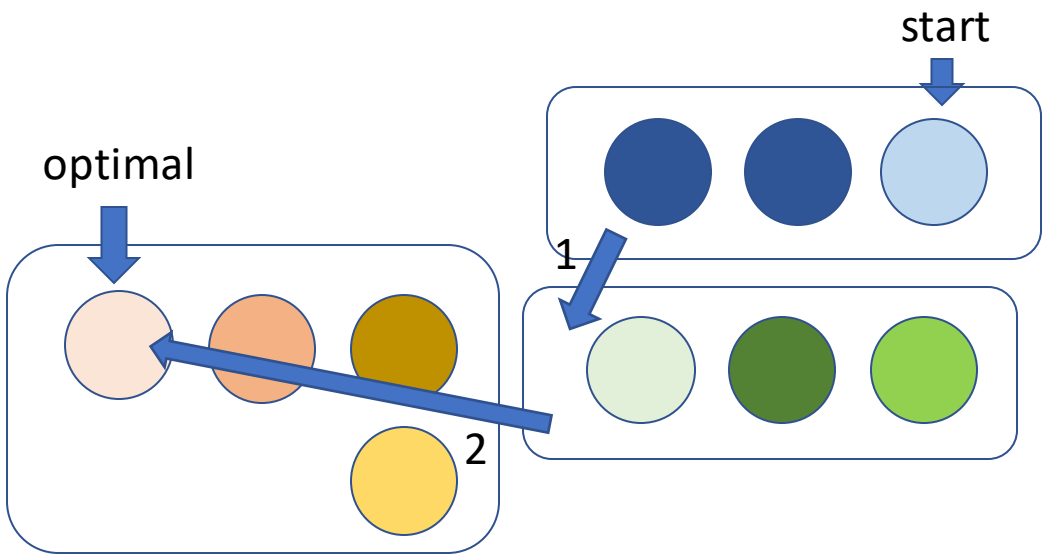
- limit the depth of primitives combination
  - (e.g., a single loop can use split and fuse recursively)
- prune the parameter space
  - split factors to divisible
- pre-determine certain decisions for different hardware
  - CPU=> only parallelize the outer-most loop (after fusion) and vectorize the inner-most loop
  - GPU => bind outer loops to blocks and inner loops to threads
  - FPGA => 3 stage pipeline design

Schedule Space Rearrangement



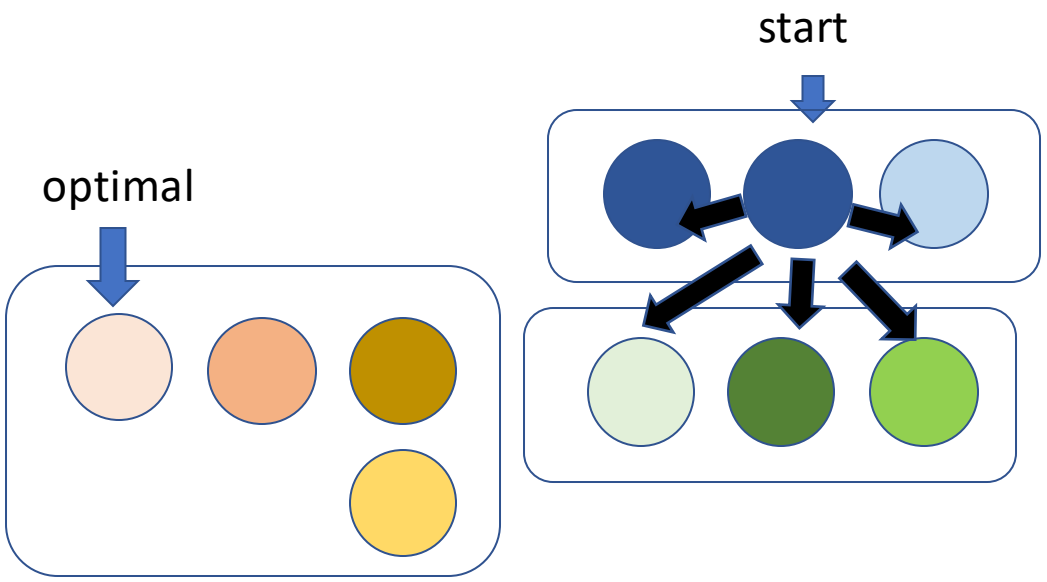
Schedule Space Rearrangement





rearrange to high dimensional space can potentially **shorten the path from starting point to optimal point**

Effective Exploration



Which point to start with?

- Heuristics: simulated annealing

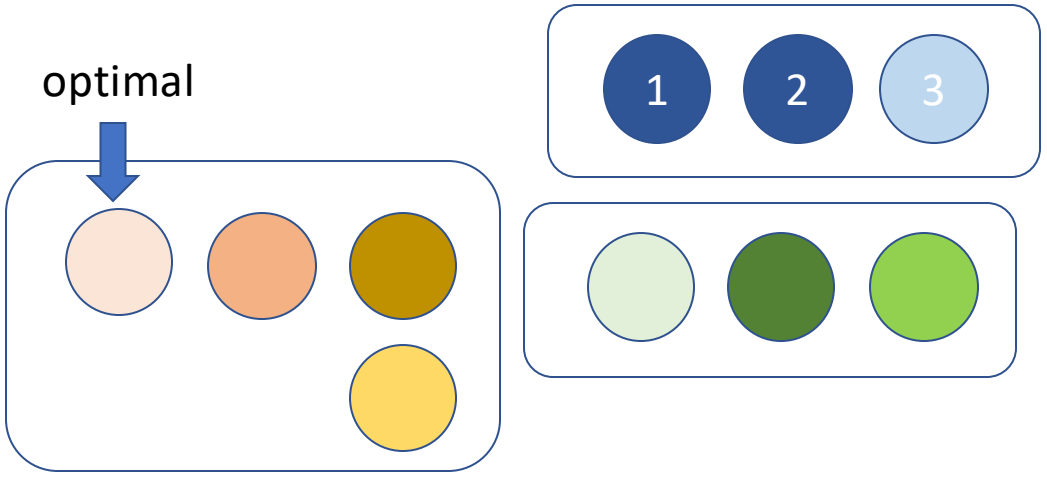
Which direction to search along?

- Machine Learning: QLearning

How to evaluate each point?

- Run on target device
- Cost model

Heuristics: simulated annealing



choose from 1, 2, and 3 known value: v^1, v^2 , v^3

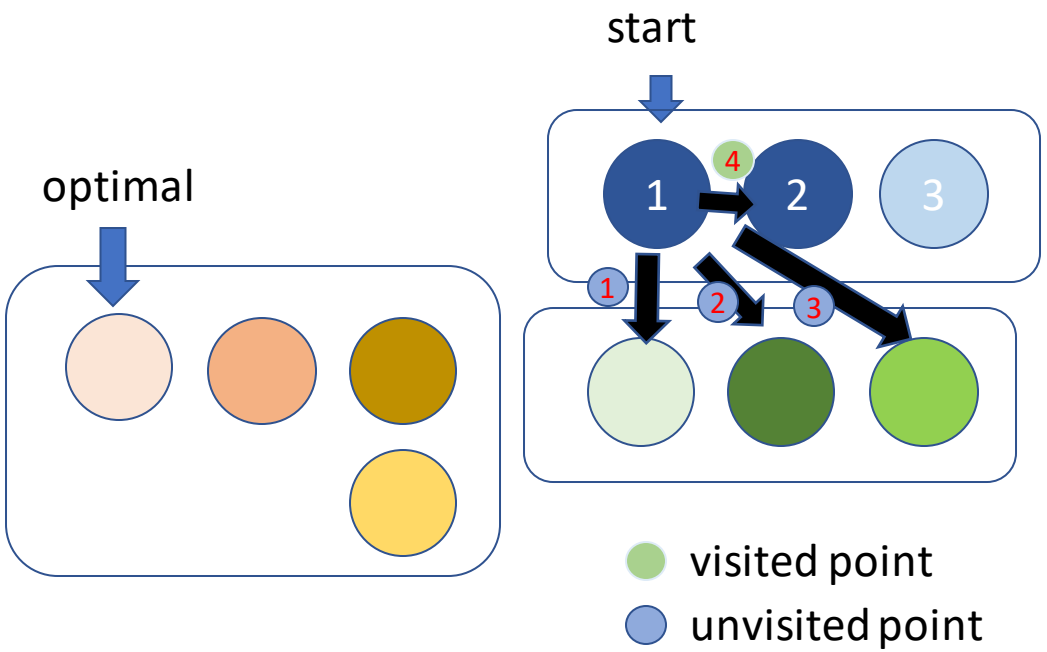
the best one known: v^*

choose according to possibility:

$$e^{-\gamma\frac{(v^*-v^i)}{v^*}}, i = 1,2,3$$

allow choosing multiple points

Machine Learning: Q-Learning



1. keep record of visited points: discard 4
2. use DQN algorithm to predict Q-value of each direction q^1, q^2, q^3
3. choose the largest one: $q^* = \max(q^i), i = 1,2,3$

MDP

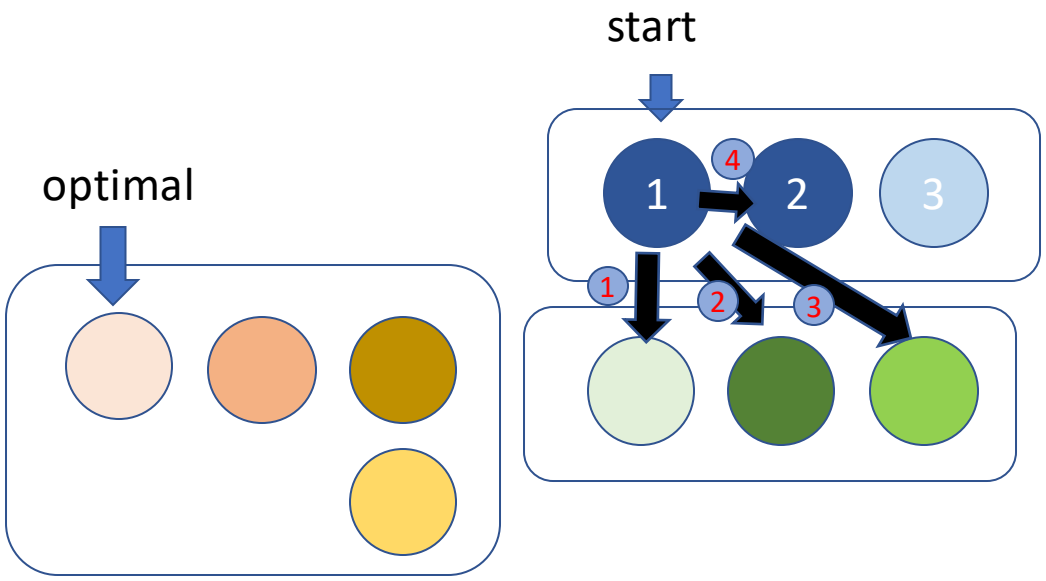
State : point p

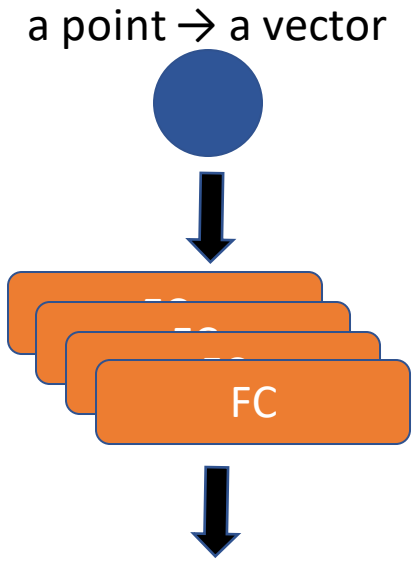
Action : direction ds

Reward : $Ee - Ep/Ep$

(where E is performance value)

DQN Algorithm





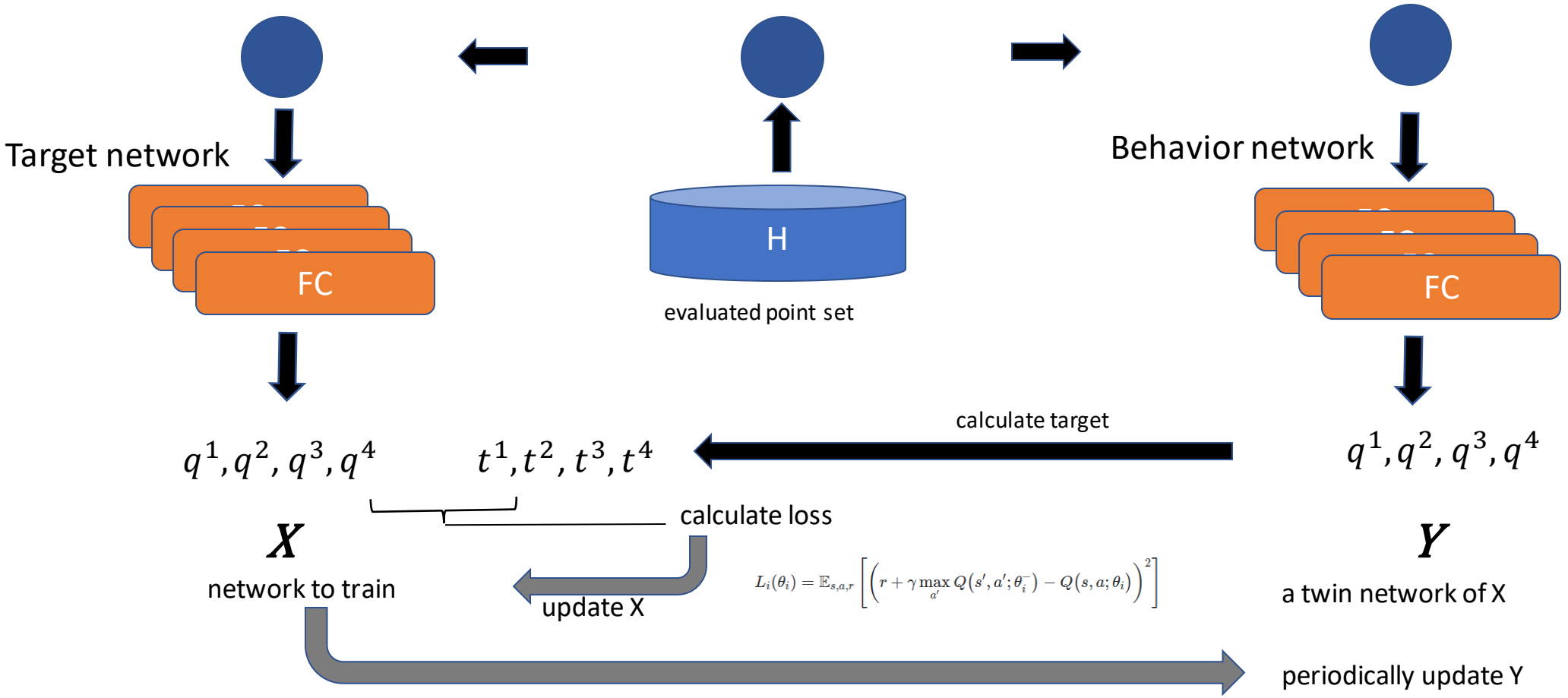
q^1, q^2, q^3, q^4

discard q^4

$q^* = \max(q^i), i = 1,2,3$

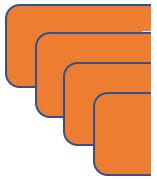
fixed number of directions (because of fixed dimension)

DQN: Train


Target network
FC
H
evaluated point set
Behavior network
FC
calculate target
q^1, q^2, q^3, q^4
t^1, t^2, t^3, t^4
q^1, q^2, q^3, q^4
calculate loss
X
network to train
update X
$L_i(\theta_i) = \mathbb{E}_{s,a,r}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)\right)^2\right]$
Y
a twin network of X
periodically update Y

DQN: Train

Target network



$q^1, ($

net

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

periodically update Y

Performance Comparison(reward)

performance value

- Measure the real performance
  - easy to implement
  - Portable
  - long time to complete
- Use analytical model
  - very fast
  - non-trivial to build

CPU &GPU

- Measure the real performance

FPGA

- $Execution\_time = workload \times \max(R, C, W) / \#PE$
  - R is the data read time
  - W is the data write time
  - C is the computation time
  - $\#PE$ is # of parallel processing elements
  - the longest stage in the pipeline x $\#PE$

Experiment: Methodology

Benchmarks

- 12 widely used operators (Conv, GEMM, etc)
- Each operator with several (5-15) test cases

Evaluation

- Geometric mean speedup
- Absolute performance

Baseline

- CPU: MKL-DNN
- GPU: cuDNN
- FPGA: hand-optimized

Experiment: GPUs

Table 3. Benchmark specifications.

Tensor Computations		Analysis Results		Library Support		FLOPs	Precision	Test Cases
Operator	Abbr.	#sl/rl	#node	CPU	GPU			
GEMV	GMV	1/1	1	MKL	cuBlas	16K-1M	float32	6
GEMM	GMM	2/1	1	MKL	cuBlas	32K-8.6G	float32	7
Bilinear	BIL	2/2	1	MKL	cuBlas	1G	float32	5
1D convolution	C1D	6/2	2	MKL-DNN	cuDNN	50M-200M	float32	7
Transposed 1D convolution	T1D	9/2	3	PyTorch	cuDNN	50M-200M	float32	7
2D convolution	C2D	8/3	2	MKL-DNN	cuDNN	77M-3.7G	float32	15
Transposed 2D convolution	T2D	12/3	3	PyTorch	cuDNN	77M-3.7G	float32	15
3D convolution	C3D	10/4	2	PyTorch	cuDNN	77M-6.6G	float32	8
Transposed 3D convolution	T3D	15/4	3	PyTorch	cuDNN	77M-6.6G	float32	8
Group convolution	GRP	4/3	2	MKL-DNN	cuDNN	20M-900M	float32	14
Depthwise convolution	DEP	4/3	2	MKL-DNN	cuDNN	250K-3.6M	float32	7
Dilated convolution	DIL	4/3	2	MKL-DNN	cuDNN	100M-1.2G	float32	11

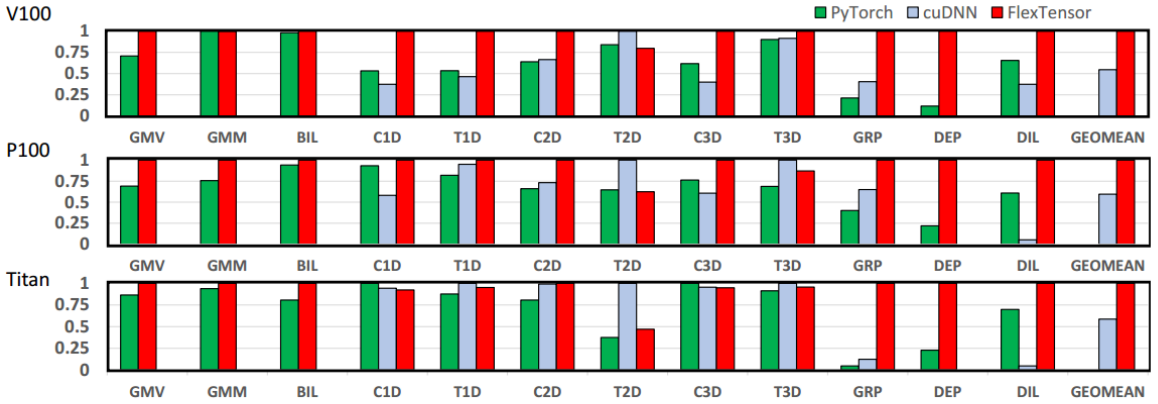


Figure 5. Normalized performance of native PyTorch, cuDNN and FlexTensor on different GPUs.

- Three GPU platforms
- Speedup over cuDNN
  - 1.83x on V100
  - 1.68x on P100
  - 1.71x on Titan X

Experiment: Conv2d

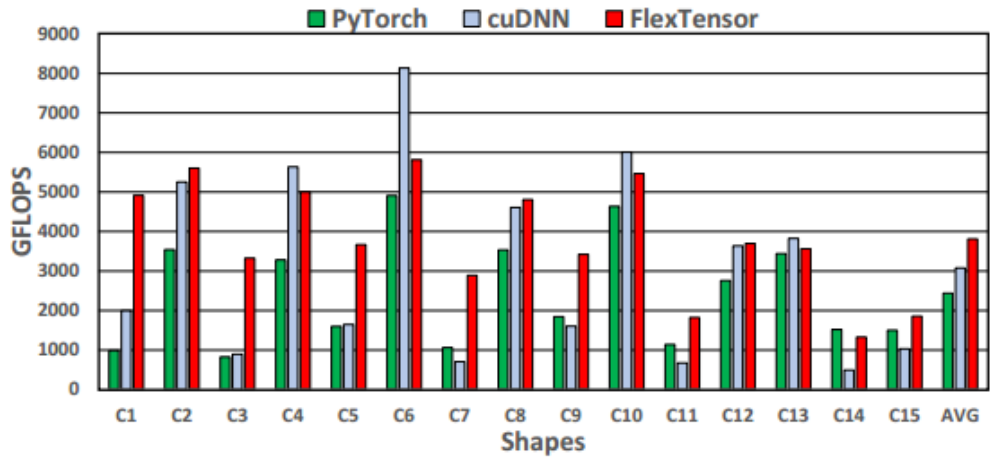
Table 4. Configurations of 15 distinctive convolution layers in YOLO v1.

Name	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15
C	3	64	192	128	256	256	512	256	512	512	1024	512	1024	1024	1024
K	64	192	128	256	256	512	256	512	512	1024	512	1024	1024	1024	1024
H/W	448	112	56	56	56	56	28	28	28	28	14	14	14	14	7
k, st	7,2	3,1	1,1	3,1	1,1	3,1	1,1	3,1	1,1	3,1	1,1	3,1	3,1	3,2	3,1

Test on heterogeneous system, including GPU, CPU, and FPGA

- Codegen for GPU: generate CUDA
- Codegen for CPU: generate LLVM IR
- Codegen for FPGA: generate OpenCL HLS

Experiment: Conv2d-GPU



(a) Results of FlexTensor compared to PyTorch (without cuDNN) and cuDNN library on NVIDIA V100 GPU for 2D convolutions.

Speedup (Geometric mean):

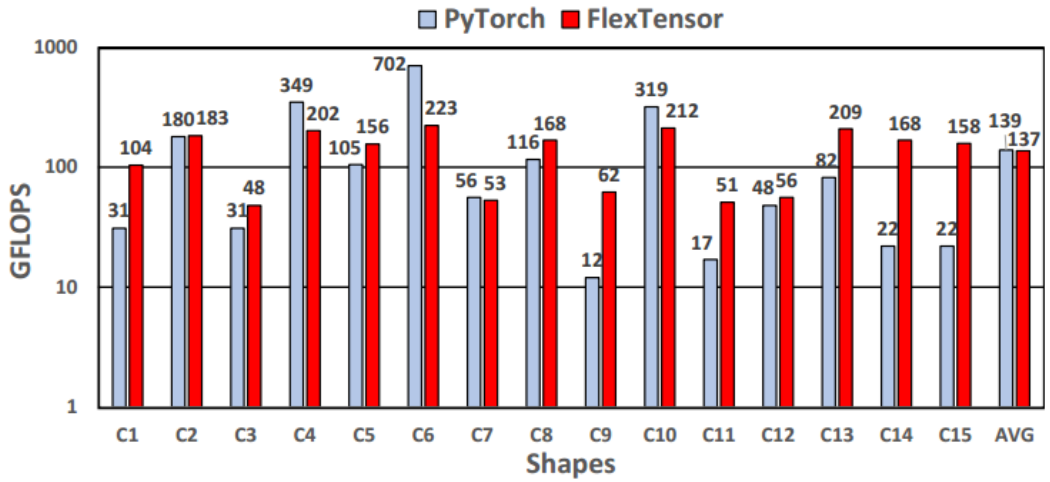
- **1.5x** speedup over cuDNN
- **1.56x** speedup over PyTorch (native)

Absolute performance (average):

- **PyTorch**: 2438.74 GFLOPS
- **cuDNN**: 3076.70 GFLOPS
- **FlexTensor**: 3810.96 GFLOPS

C4 and C6 =>Winograd algorithm

Experiment: Conv2d-CPU



(b) Results of FlexTensor compared to PyTorch on Intel Xeon E5-2699 v4 for 2D convolutions.

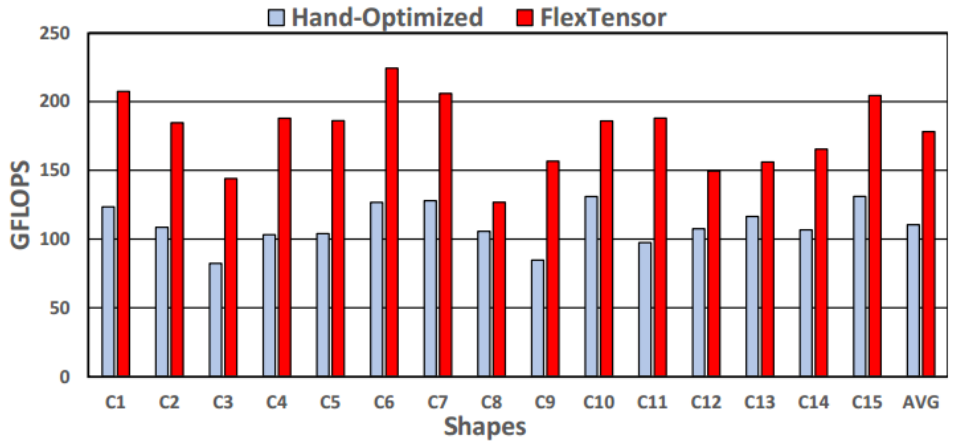
Speedup (Geometric mean):

- **1.72x** to MKL-DNN

Absolute performance (average):

- **MKL-DNN**: 139.49 GFLOPS
- **FlexTensor**: 136.91 GFLOPS

Experiment: Conv2d-FPGA



(c) Results of FlexTensor comapred to hand-optimized OpenCL baselines on Xilinx VU9P FPGA for 2D convolutions.

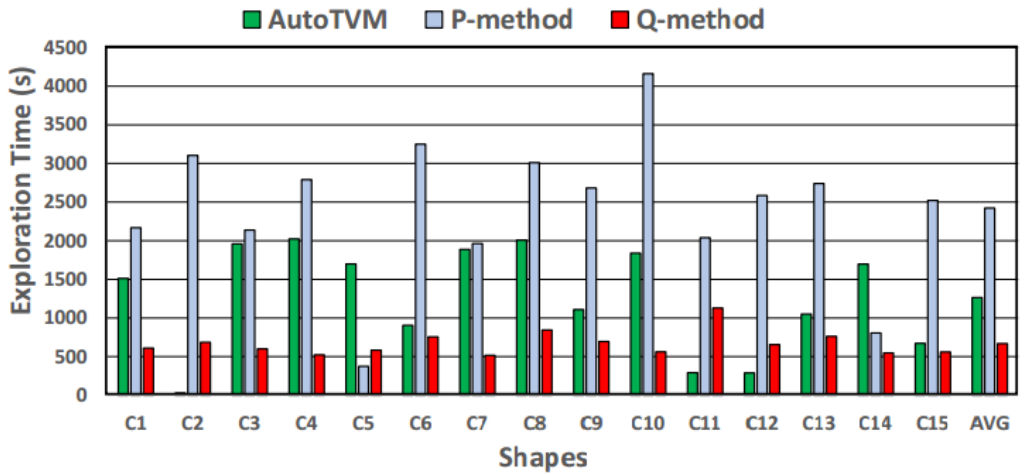
Speedup (Geometric mean):

- **1.5x** speedup over hand-opt

Absolute performance (average):

- **Hand-opt**: 110.42 GFLOPS
- **FlexTensor**: 178.16 GFLOPS

Compare to State-of-the-art



(d) Exploration time comparison of AutoTVM, P-method, and Q-method.

AutoTVM:

- Auto-tuning tool for TVM
- High-performance
- Semi-automatic
- Requires template
- Uses Xgboost/treeRNN to learn a cost model

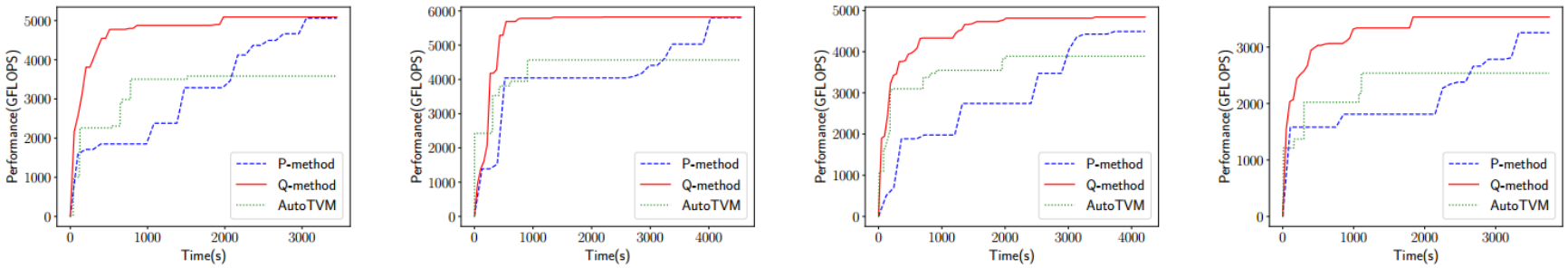
Benchmarks:

- Different kinds of convolutions
- Overall speedup **2.21x(table3)**
- P-method is 1.41x better than AutoTVM
- Q-method is 1.54x better than AutoTVM

Methods:

- Q-method : Q-learning based method
- P-method : search and tries all
- possible directions

Compare to State-of-the-art



(a) C1 (b) C6 (c) C8 (d) C9

Figure 7. Performance vs. Exploration time

Q-method (red): our method
P-method (blue): ours without Q-learning
Baseline (green): AutoTVM
Q-method only use **27.6**% the time of AutoTVM

Compare to State-of-the-art(AutoTVM)

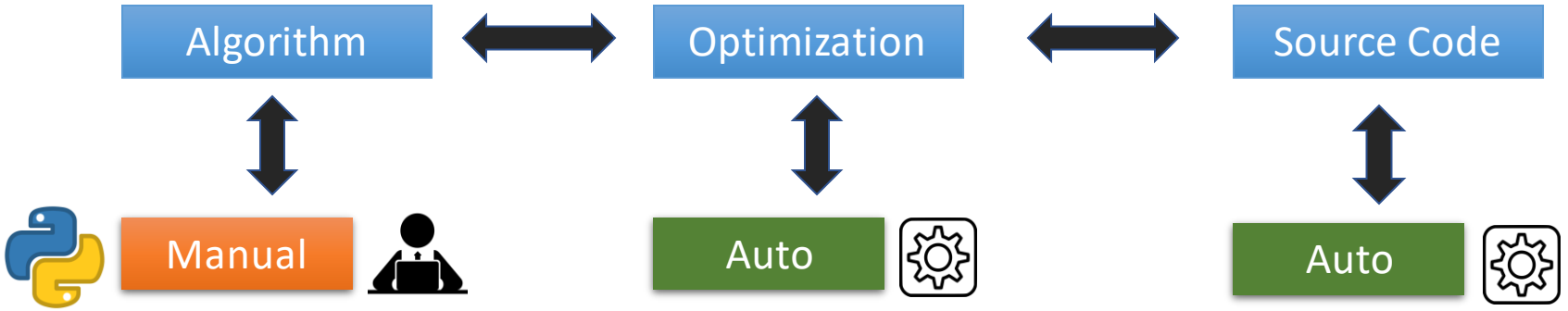
New Operators

- Block-circulant-matrix: **2.11x** speedup
- Shift-convolution: **1.53x** speedup
- Fully automatic

Whole networks

- Overfeat: **1.39x** speedup
- YOLO-v1: **1.07x** speedup
- With graph optimization

Conclusion



- **Fully automatic** scheduling exploration
- **Heterogeneous system** : CPU, GPU, FPGA
- **Heuristics and Q-learning**
- **High performance**.