SIRIUS: Harvesting Whole-program Optimization Opportunities for DNNs

Yijin Li*+^ Jiacheng Zhao*+#^ Qianqi Sun*+ Haohui Mai & Lei Chen*+ Wanlu Cao*+ Yanfan Chen*+ Zhicheng Li*+
Ying Liu*# XinYuan Zhang* Xiyu Shi* Jie Zhao† Jingling Xue† Huimin Cui*+ Xiaobing Feng*+

SKLP, Institute of Computing Technology, CAS*

University of Chinese Academy of Sciences+

HengMuXing Technologies&

State Key Laboratory of Mathematical Engineering and Advanced Computing†

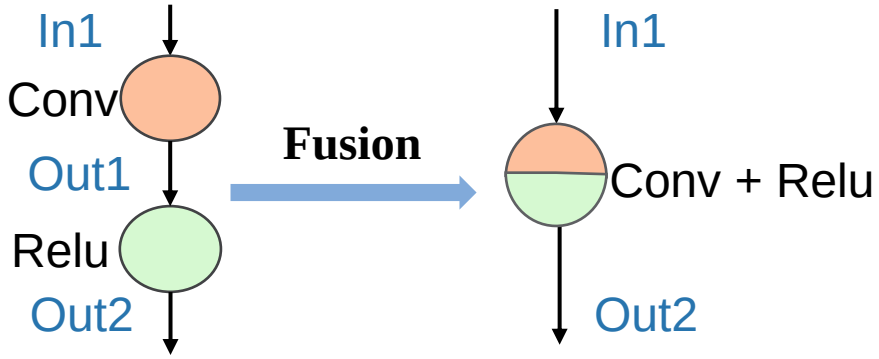
The University of New South Wales†

ZhongGuanCun Laboratory#

Equal Contribution^

Fusion is key for optimizing accelerator program

- Widely adopted in deep learning compiler and frameworks
  - Fuse producer-consumer operators by data reuse
  - Improve accelerator performance:
    - Reduced kernel launch costs (~2.5μs)
    - Minimizing data traffic to off-chip memory (1.55 TB/s for global vs 54 TB/s for shared)



An example for fusion: Conv + Relu

Memory Unit	Bandwidth (TB/s)
Global	1.55
L1	54
L2	4.8
Shared	54

Bandwidth of NVIDIA A100

Fusion in deep learning compiler/framework

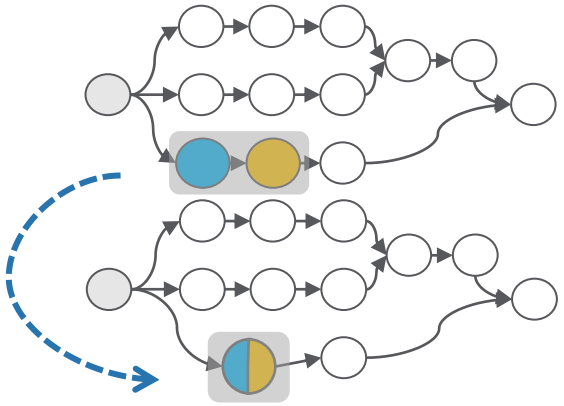
Computational Graph

High Level Graph Rewriting

Optimized Computational Graph

Operator-level Optimization

Tensor Expressions



```
for i, j in T.grid(I, J):
    for k in range(K):
        C[i, j] += A[i, k] * B[k, j] # matmul block
    compute[i, j] = T.max(C[i, j], 0) # relu block
```

Graph level

Apollo[MLSys'22]
AStitch[ASPLOS'21]
PET[OSDI'21]
DNNFusion[PLDI'21]
Rammer[OSDI'20]
TASO[OSDI'20]

AI compiler

Native compiler

C++ CUDA …

Compiler

LLVM IR

Backend

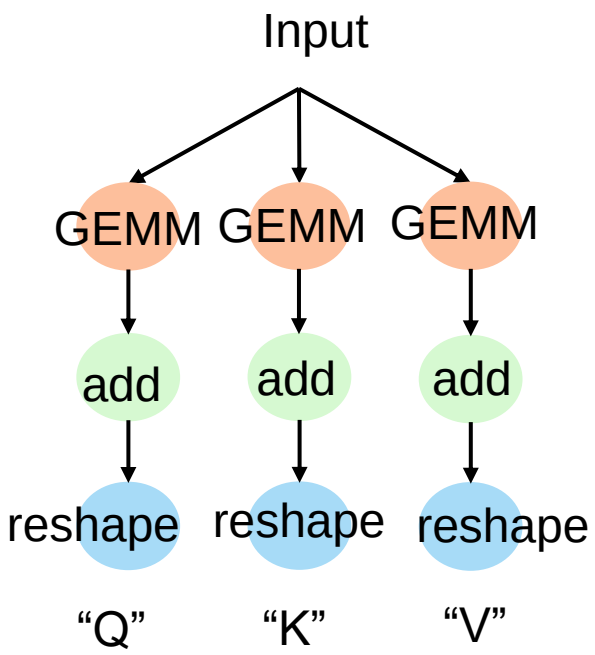
Hardware

Launch Kernel 1
Launch Kernel 2

```
__global__ void kernel(…) {
  …
}
```

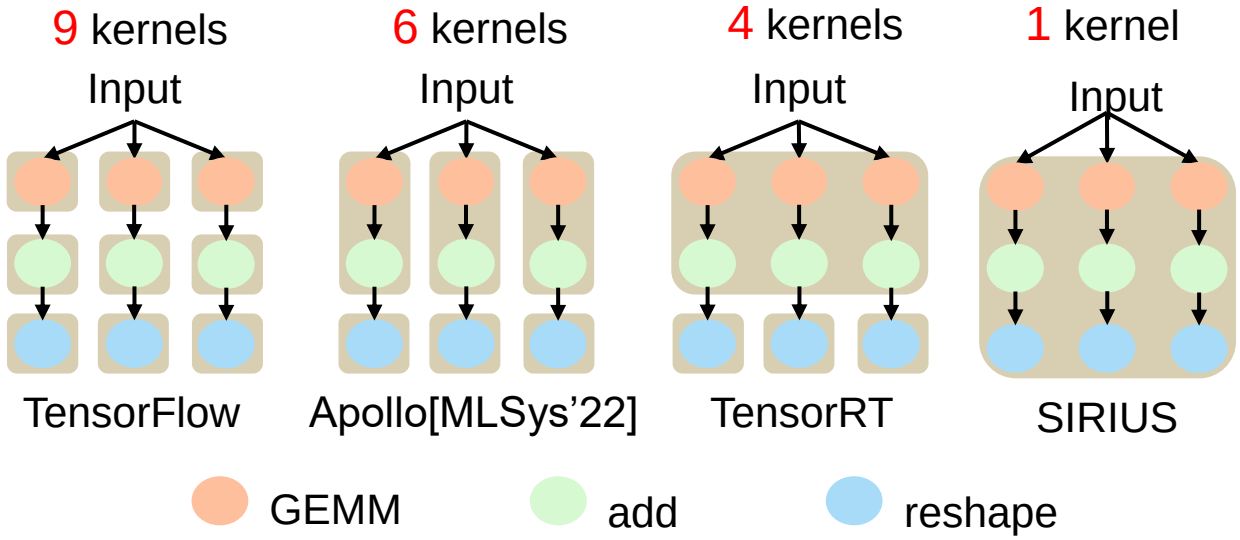
C++ and CUDA code

STOA fusion work miss certain fusion opportunities



The sub-graph of BERT

Fusion



Fusion Results for the sub-graph of BERT

	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

Performance characteristics

SIRIUS optimizes whole-program from the bottom

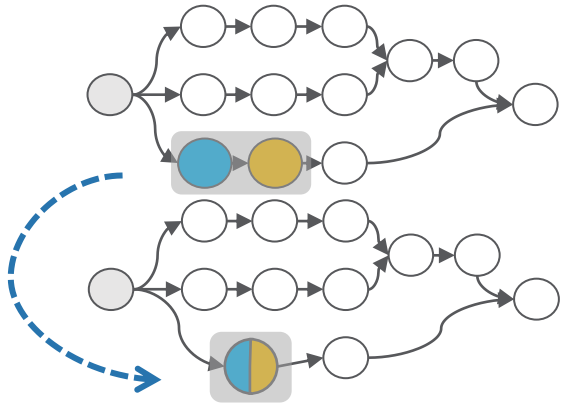
Computational Graph

High Level Graph Rewriting

Optimized Computational Graph

Operator-level Optimization

Tensor Expressions



```
for i, j in T.grid(I, J):
    for k in range(K):
        C[i, j] += A[i, k] * B[k, j] # matmul block
    compute[i, j] = T.max(C[i, j], 0) # relu block
```

Graph level

Apollo[MLSys'22]
AStitch[ASPLOS'21]
PET[OSDI'21]
DNNFusion[PLDI'21]
Rammer[OSDI'20]
TASO[OSDI'20]

AI compiler

C++ CUDA …

Compiler

LLVM IR

Backend

Hardware

Launch Kernel 1
Launch Kernel 2

```
__global__ void kernel(…) {
  …
}
```

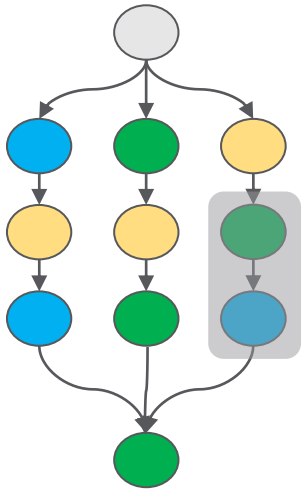
C++ and CUDA code

Native compiler

SIRIUS

Challenge #1: Design IR to facilitate dependence

- Dependency is key for fusion
- Operator dependence is on data flow graph
- Kernel dependence is non-trivial on source code



one-to-one: relu

one-to-many: resize

many-to-many: conv

Data flow graph

C++ code

```
int main(){
    ...
    // Kernel launch with N threads
    Conv<<<M, N>>>(Out, in, weight);
    ...
    Relu<<<M, N>>>(Out, OUt);
}
```

CUDA code

```
// Kernel definition
__global__ void Conv(float* A, float* B, float* C) {
   int index = threadIdx.x + …;
   A[index] = …;
}
__global__ void Relu(float* D, float* C) {
   int index = threadIdx.x + …;
   … = C[index];
}
```

Source code

Challenge #2 Fusion and code generation on low level IR

Fuse schedule for kernels

Can be fused?

C++ code

```
int main(){
    ...
    // Kernel launch with N threads
    Conv<<<M, N>>>(Out, in, weight);
    ...
    Relu<<<M, N>>>(Out, OUt);
}
```

CUDA code

```
// Kernel definition
__global__ void Conv(float* A, float* B, float* C) {
   int index = threadIdx.x + …;
   A[index] = …;
}
__global__ void Relu(float* D, float* C) {
   int index = threadIdx.x + …;
   … = C[index];
}
```

Generate code for kernels on low level IR

```
// Conv kernel
define void Conv(…,float addrspace(1)* %C) {
 …
 store float %valC, float addrspace(1)* %ptrC, align 4
}
```

Can fuse

+

```
// Relu Kernel
define void Relu(…,float addrspace(1)* %C) {
        …
 %valC = load float, float addrspace(1)* %ptrC, align 4
 %cmp = icmp eq i32 %valC, %y
        …
}
```

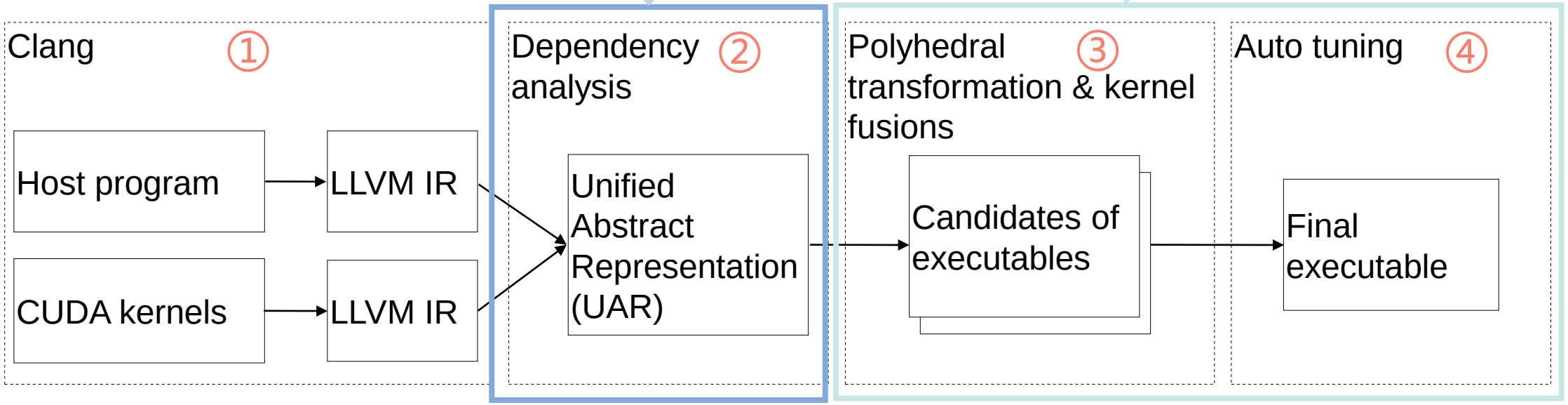
Automatic code generation

```
// Conv_Relu Kernel
define void Relu(…,float addrspace(1)* %C) {
        …
  %cmp = icmp eq i32 %valC, %y
        …
}
```

SIRIUS: Whole-program Optimizer for DNNs

1. Whole-program Representation
How to represent source code for fusion?

2. Whole-program Optimization
How to optimize fusion and code generation?



Overall Architecture for SIRIUS

UAR: modeling whole-program using polyhedral representation

Key point: model CUDA parallelism as explicit loop nest

```
// Kernel definition
__global__ void VecAdd(float* A, float* B,
float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```

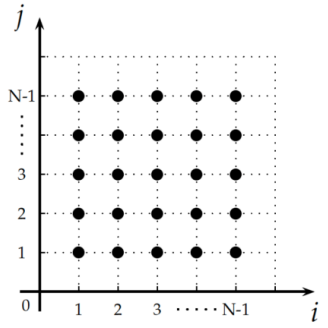
CUDA code

```
void add(float* in1, float* in2, float* out) {
    for (int blockIdx.x = 0; blockIdx < blockDimx; blockIdx++) {
        for (int threadIdx.x = 0; threadIdx < threadDimx; threadIdx++) {
            int idx = blockIdx * threadDimx + threadIdx;
            out[idx] = in1[idx] + in2[idx];
        }
    }
}
```

The equaivalent sample code

```
for(i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
S:      a[i][j] = a[i][j-1] + a[i-1][j];
    }
}
```

A sample code



The iteration space for the sample loop

Polyhedral dependence

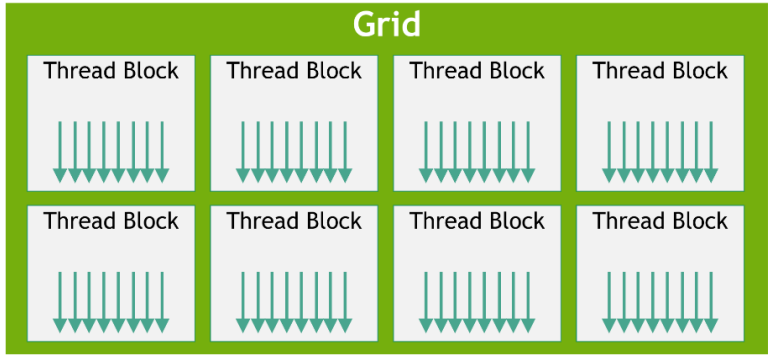
S[i][j] -> S[i][j+1] : 0 <= i <= N-1 and 0 <= j <= N-2

S[i][j] -> S[i+1][j] : 0 <= i <= N-2 and 0 <= j <= N-1

Polyhedral dependence for the sample loop

UAR: LLVM IR falls short in modeling CUDA parallelism

- CUDA kernels are executed *N* times in parallel by *N* different CUDA threads
- LLVM IR use intrinsic function to represent threads and blocks
- Intrinsic function is scalar and cannot model CUDA parallelism



CUDA parallelism: Grid of Thread Blocks.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```

A sample code for VecAdd CUDA kenel

```
define void @VecAdd(float addrspace(1)* %A, float addrspace(1)* %B, float
addrspace(1)* %C) {
entry:
  ; get threadIdx with intrinsic function.
  %idx = tail call i32 @llvm.nvvm.read.ptx.sreg.tid.x() readnone nounwind
  %ptrA = getelementptr float, float addrspace(1)* %A, i32 %idx
  %ptrB = getelementptr float, float addrspace(1)* %B, i32 %idx
  …
 }
```

LLVM IR for VecAdd CUDA kernel.

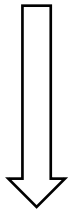
UAR: explicit loop nest to model CUDA parallelism

Use induction variable of loop to represent threadIdx and blockIdx

```
void add(half* in1, half* in2, half* out) {
    int idx = blockIdx.x * threadDimx + threadIdx.x;
    out[idx] = in1[idx] + in2[idx];
}
```

a cuda kernel sample code.

Model as loop



```
void add(half* in1, half* in2, half* out) {
    for (int blockIdx.x = 0; blockIdx < blockDimx; blockIdx++) {
        for (int threadIdx.x = 0; threadIdx < threadDimx; threadIdx++) {
            int idx = blockIdx * threadDimx + threadIdx;
            out[idx] = in1[idx] + in2[idx];
        }
    }
}
```

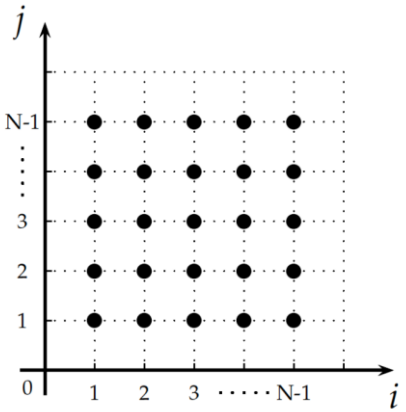
The equaivalent sample code.

UAR: a data dependency graph

- Local UAR is UAR for every function or kernel
- Global UAR is UAR for whole-program
- Loop can be unified by polyhedral model[1] to model dependence.

```
 for(i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
S:    a[i][j] = a[i][j-1] + a[i-1][j];
    }
 }
```

A sample code.



The iteration space for the sample loop.[2]

Polyhedral dependence

S[i][j] -> S[i][j+1] : 0 <= i <= N-1 and 0 <= j <= N-2

S[i][j] -> S[i+1][j] : 0 <= i <= N-2 and 0 <= j <= N-1

Polyhedral dependence for the sample loop.

[1] Zhao J, Kruse M, Cohen A. A polyhedral compilation framework for loops with dynamic data-dependent bounds[C]//Proceedings of the 27th International Conference on Compiler Construction. 2018: 14-24.

[2] Bondhugula U K. Effective automatic parallelization and locality optimization using the polyhedral model[D]. The Ohio State University, 2008.

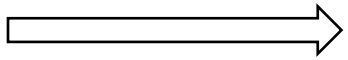
Constructing local UAR: GEMM example

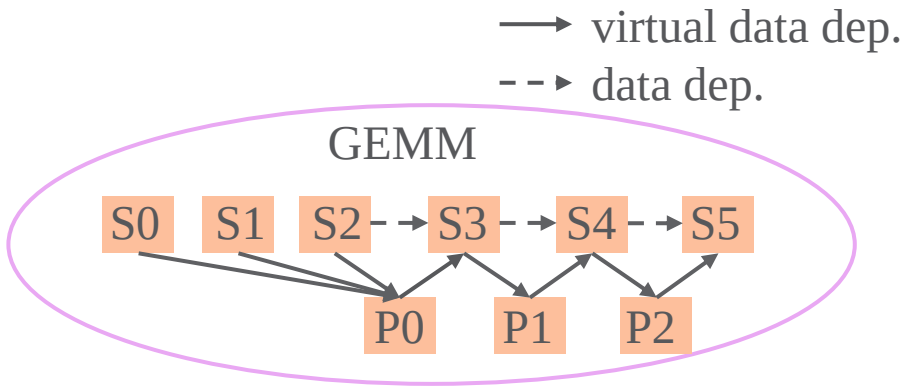
UAR is a data dependency graph

- Node is a statement
- Edge is the polyhedral dependence between statements

```
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)
}
```

Gemm CUDA kernel pseudo code

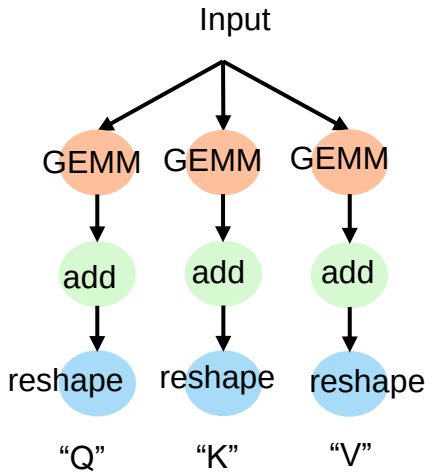




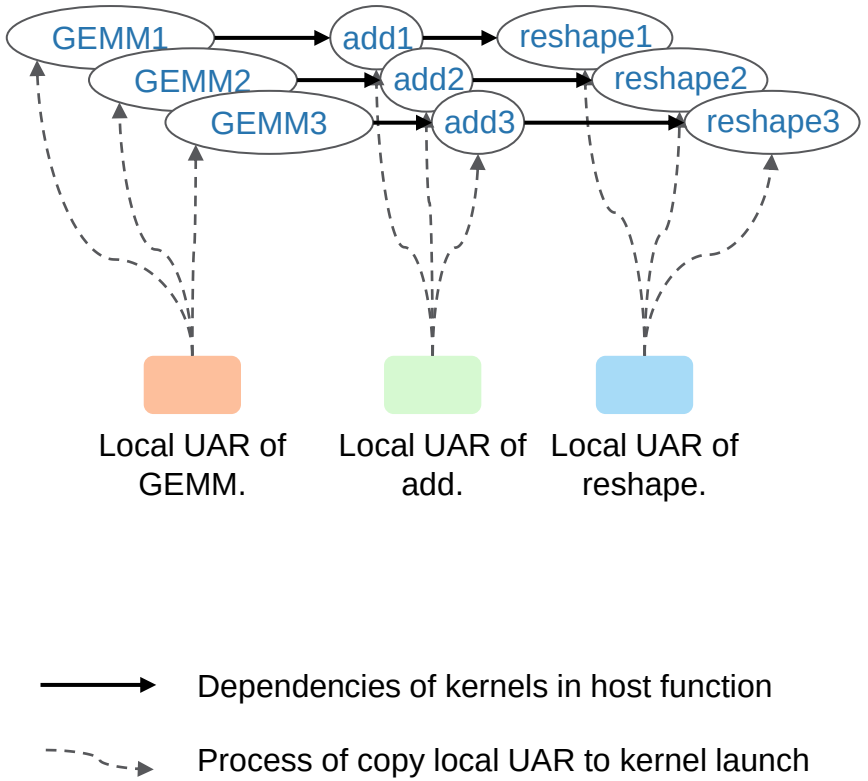
UAR of GEMM kernel

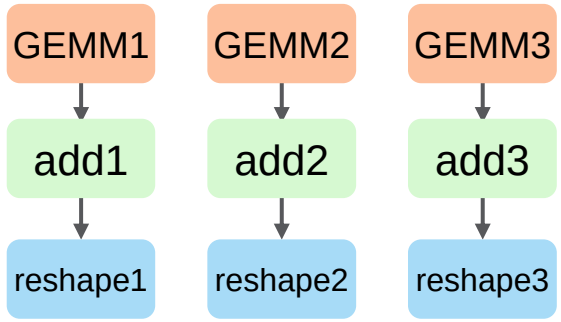
Constructing global UAR: an algorithm

Clone-based bottom-up analysis to construct global UAR




```
void bert() {
  gemm<<<96,128>>>(input, Q, outputQ)
  add<<<384,384>>>(outputQ, biasQ)
  reshape<<<384,384>>>(outputQ)
  gemm<<<96,128>>>(input, K, outputK)
  add<<<384,384>>>(outputK, biasK)
  reshape<<<384,384>>>(outputK)
  gemm<<<96,128>>>(input, V, outputV)
  add<<<384,384>>>(outputV, biasV)
  reshape<<<384,384>>>(outputV)
}
```





Computing graph and pseudo code

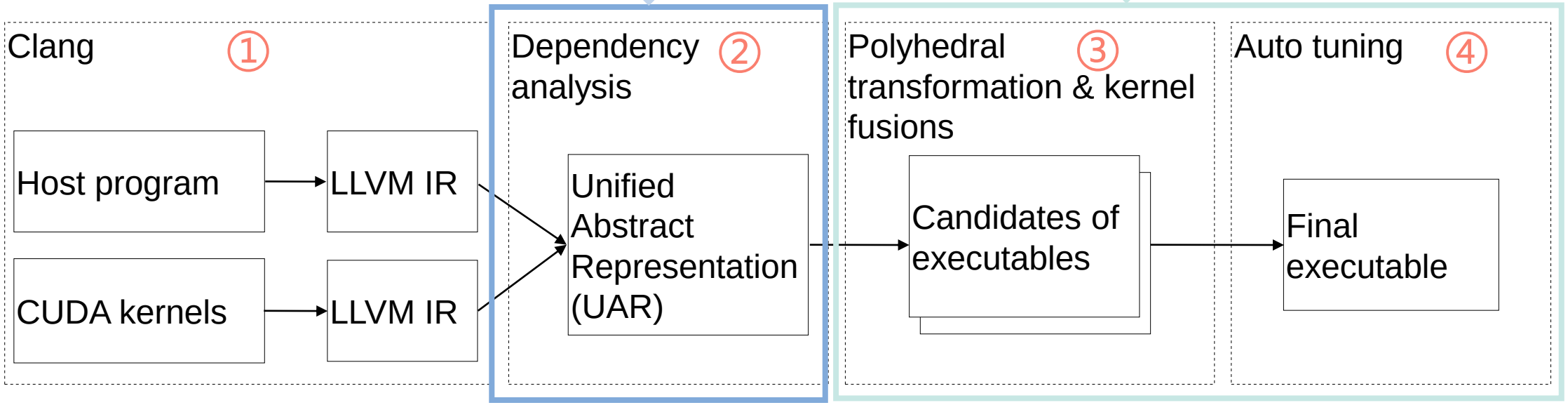
The process of solving global UAR

Global UAR of whole-program

Two challenges for fusion from whole-program

1. Whole-program Representation
How to represent source code for fusion?

2. Whole-program Optimization
How to optimize fusion and code generation?



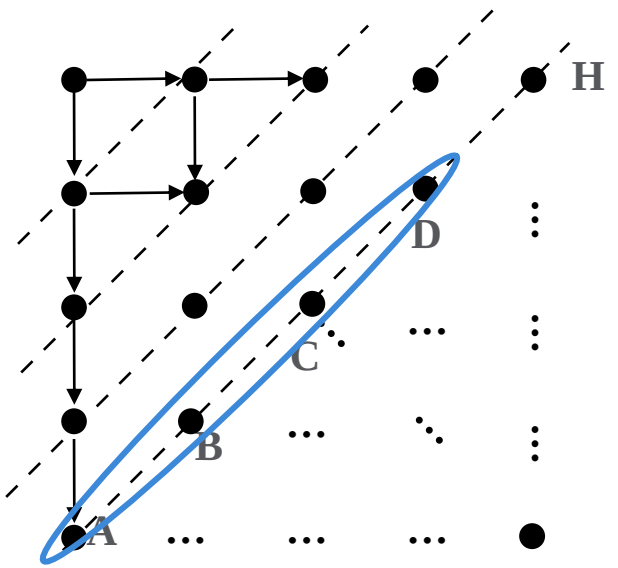
Overall Architecture for SIRIUS

Challenge #2 Fusion and code generation on low level IR

Three questions for whole-program optimization.

Solve fusion schedule

Fusable?



Fusion Schedule

Generate code

```
// Conv kernel
define void Conv(…,float addrspace(1)* %C) {
 …
 store float %valC, float addrspace(1)* %ptrC,
align 4
}
```

+

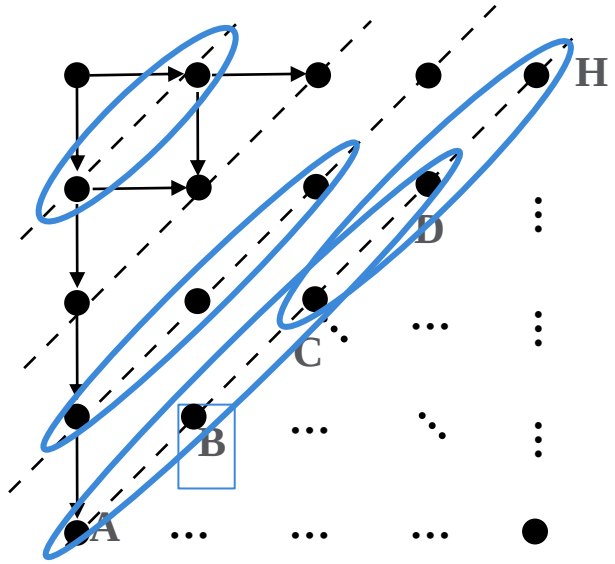
```
// Relu Kernel
define void Relu(…,float addrspace(1)* %C) {
        …
 %valC = load float, float addrspace(1)* %ptrC,
align 4
 %cmp = icmp eq i32 %valC, %y
        …
}
```

Automatic code generation

```
// Conv_Relu Kernel
__global__ void Relu(…,float addrspace(1)* %C)
{
        …
  %cmp = icmp eq i32 %valC, %y
        …}
```

Select fusion schedule

Which one is the best?



Fusion Schedule

Modeling kernel fusion as standard loop transformation

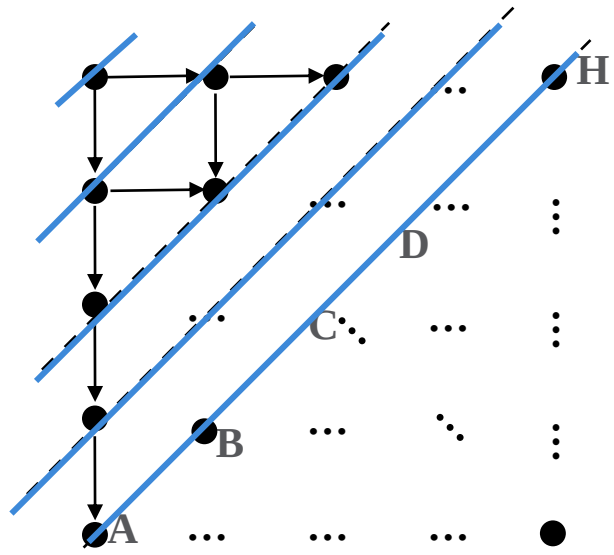
- UAR is based on polyhedral model
- ISL is an integer set library for the polyhedral Model
- Take LSTM for example: Loop in (a) is transformed to that in (b)

```
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]);
     solve(t, bias, sc[c], sh[c]);  }
```

(a) Source code for LSTM

```
void lstm() {
int wave = …;
for s in range(0, wave) {
  int minidx = …;
  int maxidx = …;
  for c in range(minidx, maxidx) {
     // 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]);
     solve(t, bias, sc[c], sh[c]); }
```

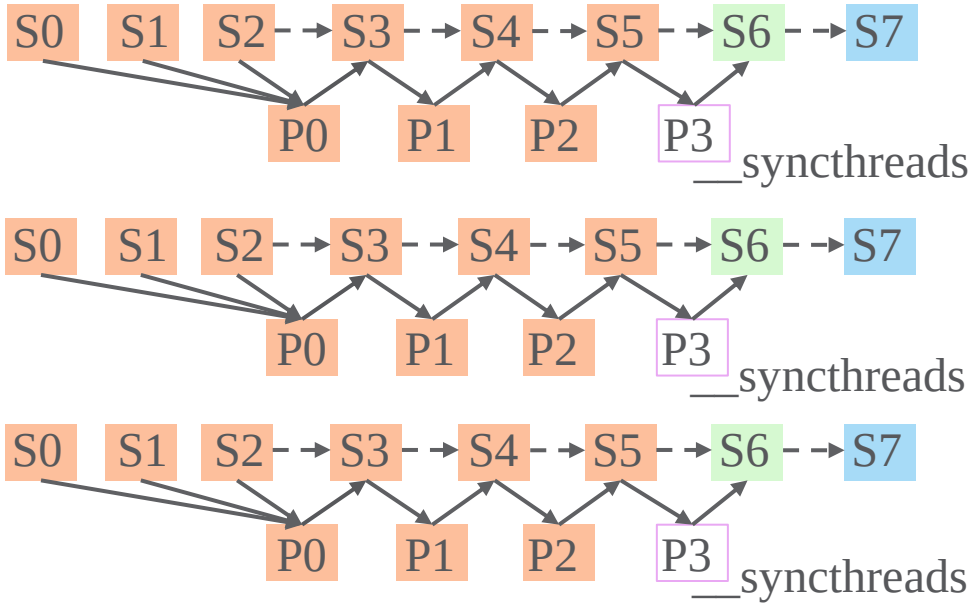
(b) Loop transformation for LSTM



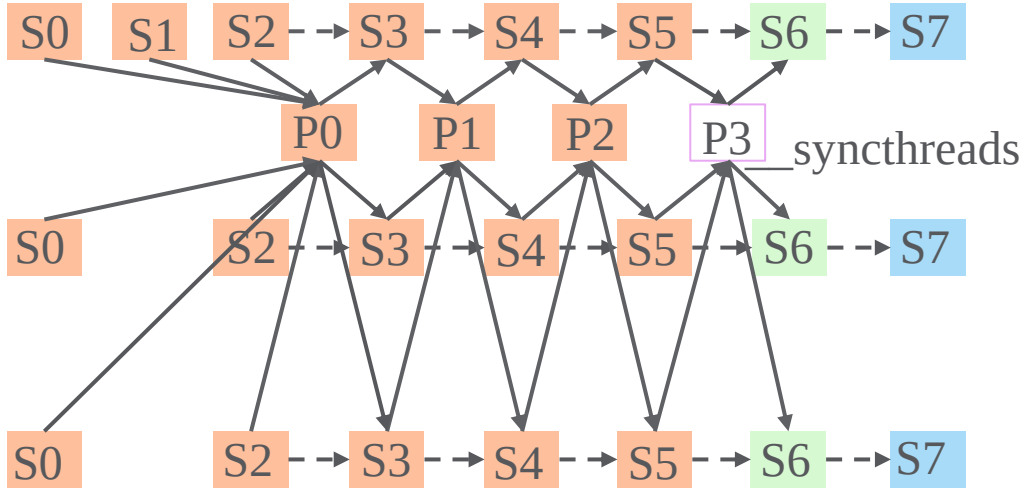
(c) Fusion schedule for LSTM

Code generation and optimization

- Collaborate with native compiler for post-fusion optimization
  - Apply CSE, Constant Propagation, redundent sync elimination

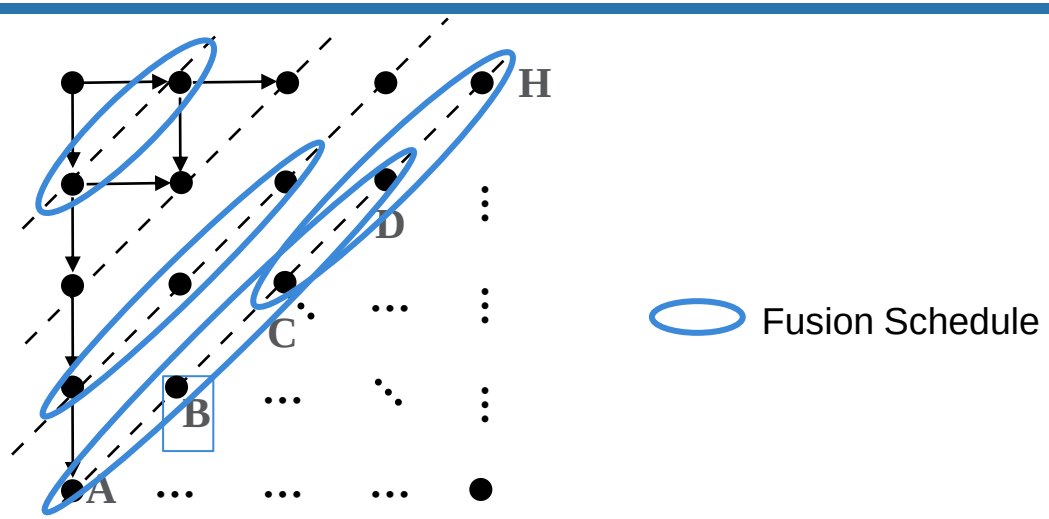


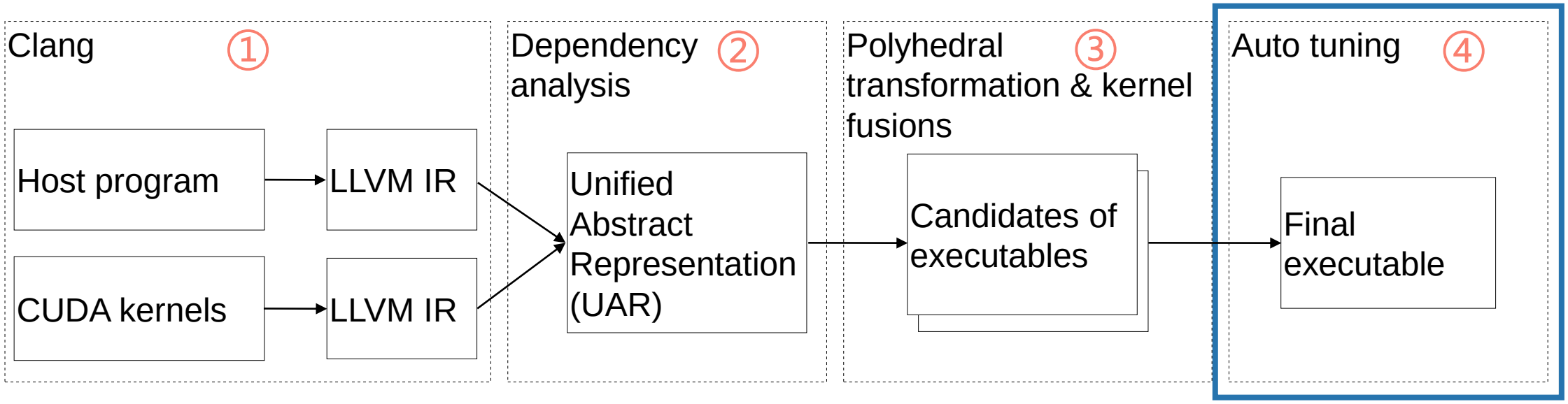
(a) Global UAR for fused kernel before opt.



(b) Global UAR for fused kernel after opt.

Auto-tune to select the best fusion schedule





Overall Arthicture for SIRIUS

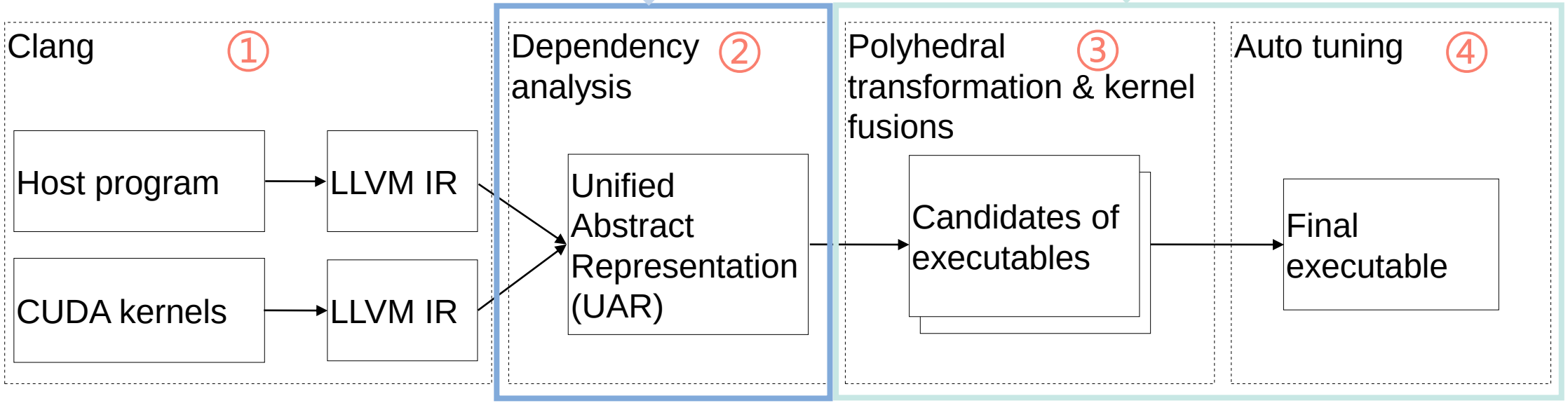
Summary

1. Whole-program Representation:

Model CUDA threads and blocks as Loop and Polyhedra
Clone-based bottom-up analysis

2. Whole-program Optimization:

Use ISL to solve fusion schedule
Generate code using post-fusion optimization
Auto-tuning to select the best fusion schedule



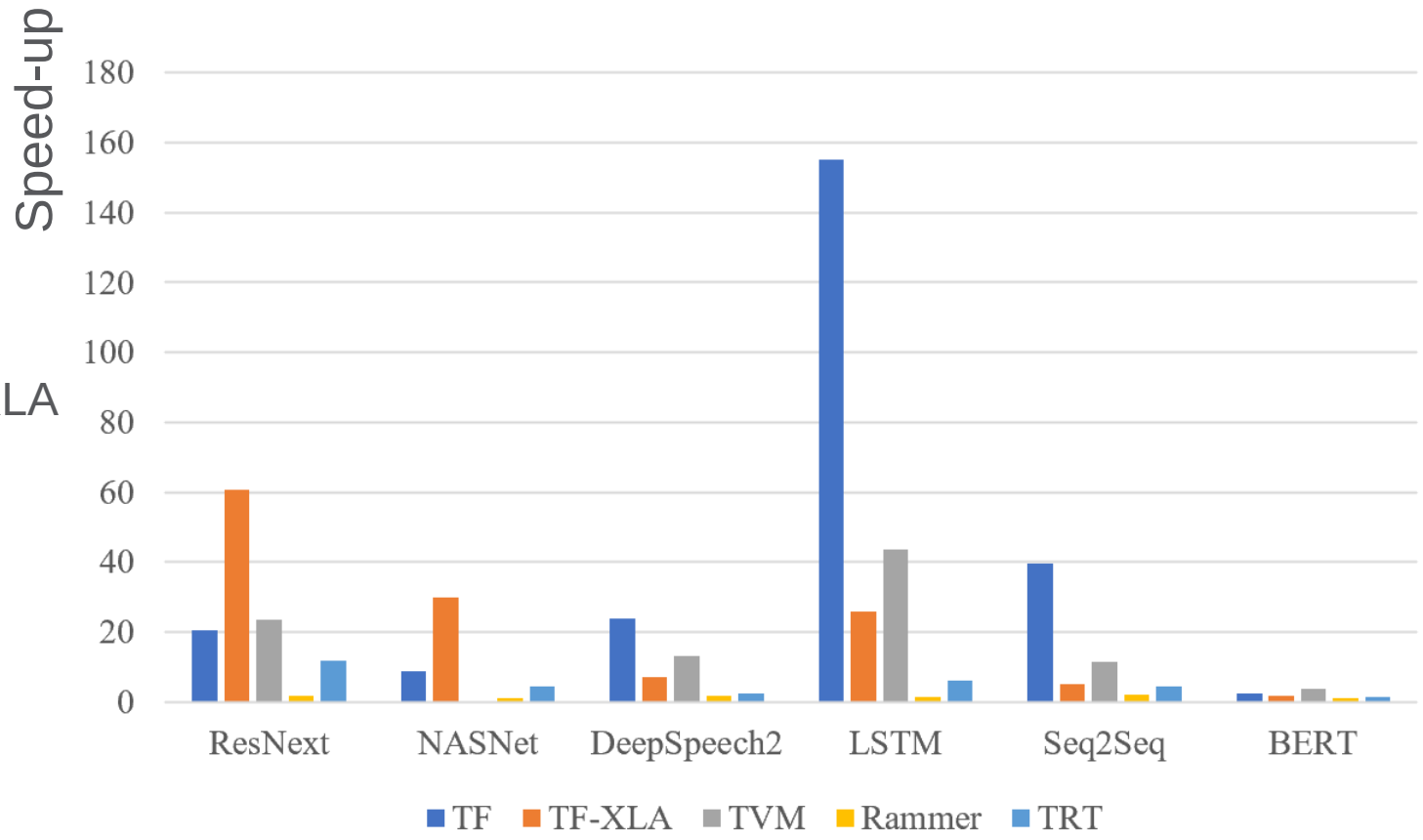
Overall Arthicture for SIRIUS

Evaluation setup

- Hardware:
  - two Intel Xeon Gold 6248 CPUs
  - 768GB of DDR4 memory
  - an NVIDIA A100 GPU
- Software:
  - Ubuntu 22.04
  - CUDA 11.8
  - TensorFlow 1.15.5
  - TensorRT 7.2

End-to-end performance on NVIDIA GPU

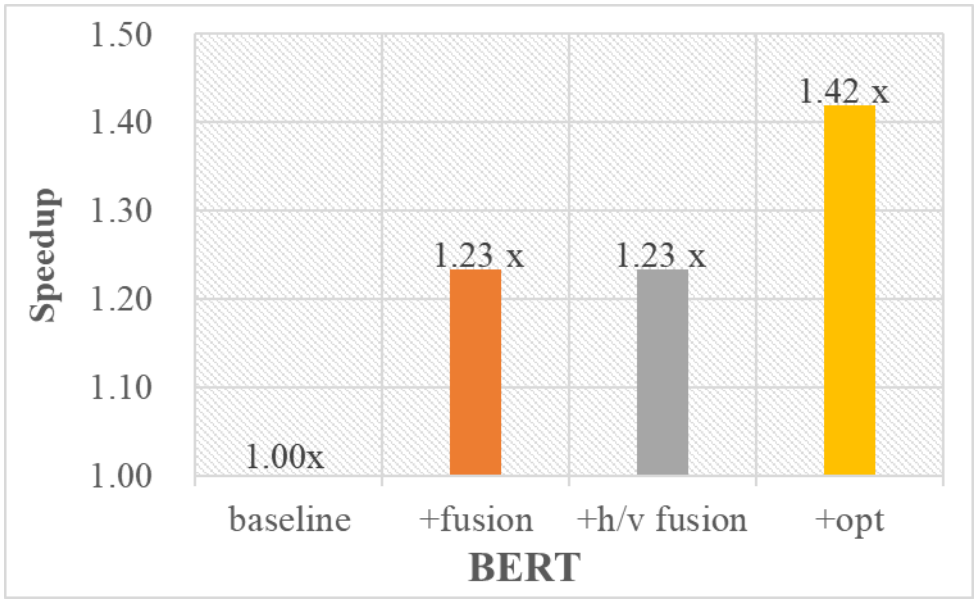
- 41.78x average speedup over TensorFlow
- 21.76x average speedup over TensorFlow-XLA
- 19.16x average speedup over TVM
- 5.15x average speedup over TensorRT
- 1.62x average speedup over Rammer

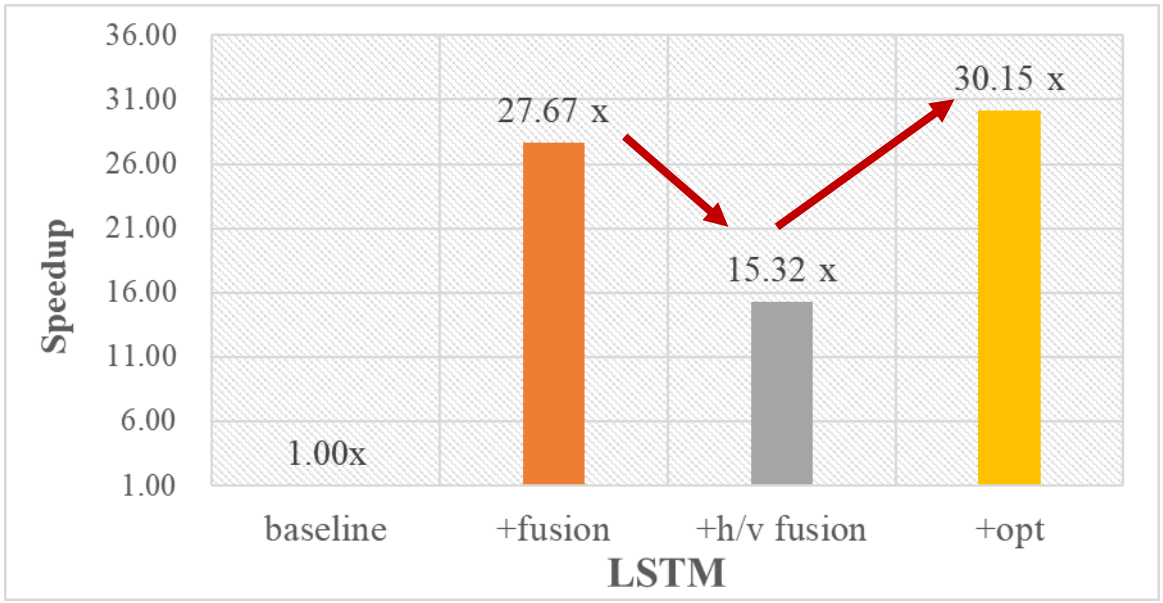


Performance Evaluation on NVIDIA A100.

Performance breakdown of SIRIUS

SIRIUS benefits from both fusion and post-fusion optimization

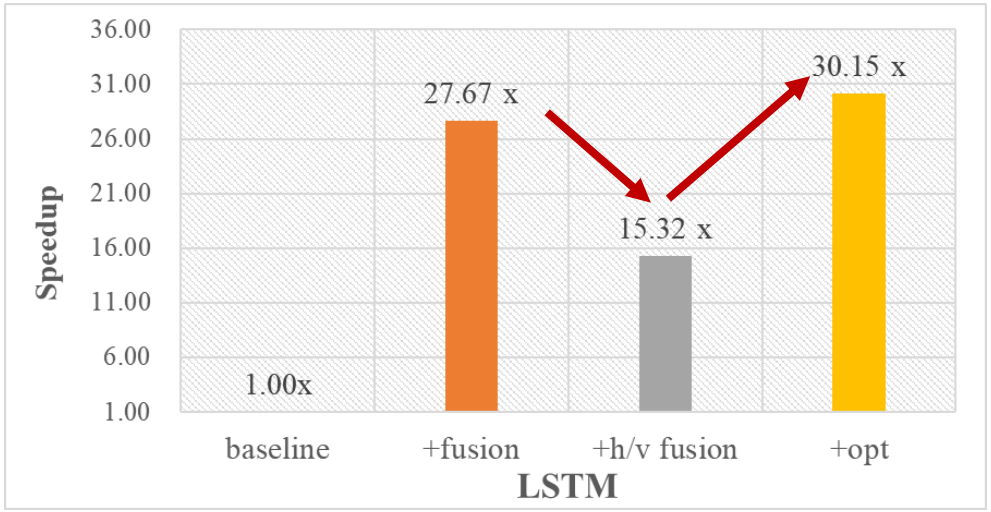




Performance breakdown for BERT and LSTM on NVIDIA A100

Case study: LSTM model

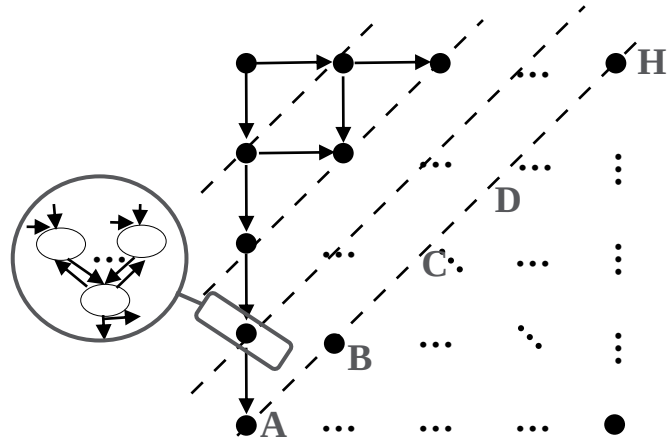
Vertical fusion is better than horizontal fusion for LSTM



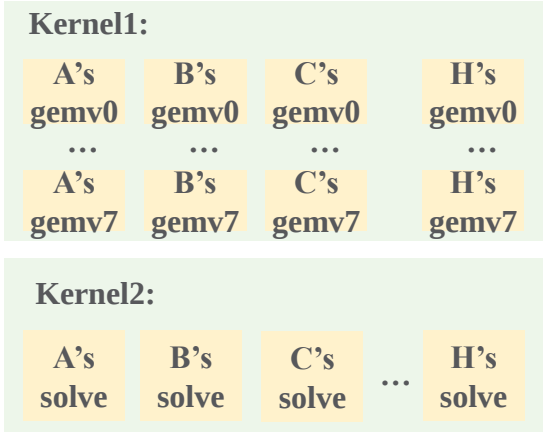
- ✓ Vetical fusion reduces parallelism
- ✓ Post-fusion optimization improves performance

```
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]);  }
```

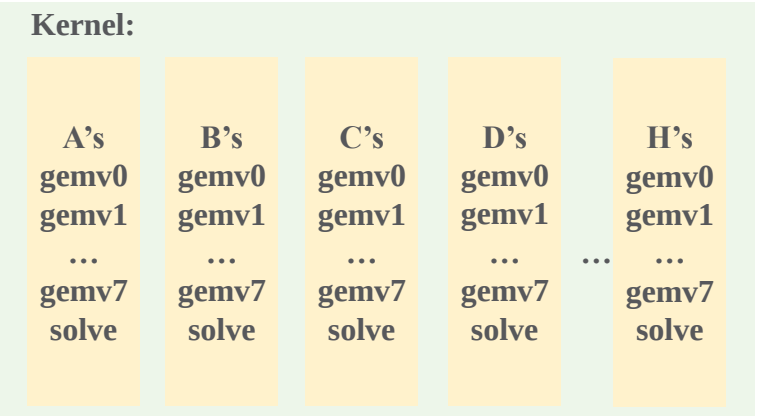
(a) Source code



(b) Fusion schedule

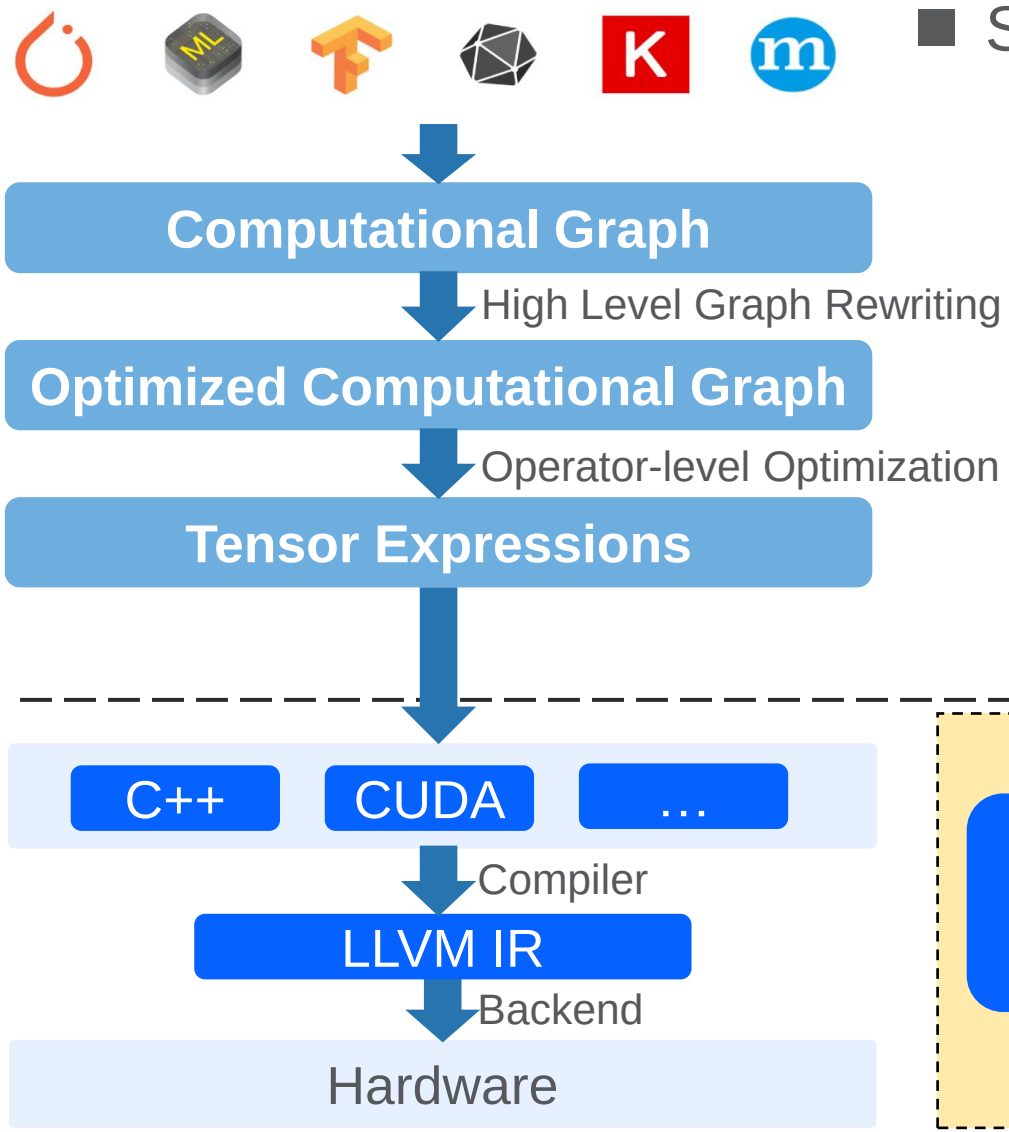


(c) Horizontal fusion (in Rammer)



(d) Vertical fusion (in SIRIUS)

Conclusion



- SIRIUS: whole-program optimizer for DNNs
  - ✓ Whole-program representation to do fusion
  - ✓ Whole-program optimization on fusion and code generation

Thanks!

liyijin@ict.ac.cn