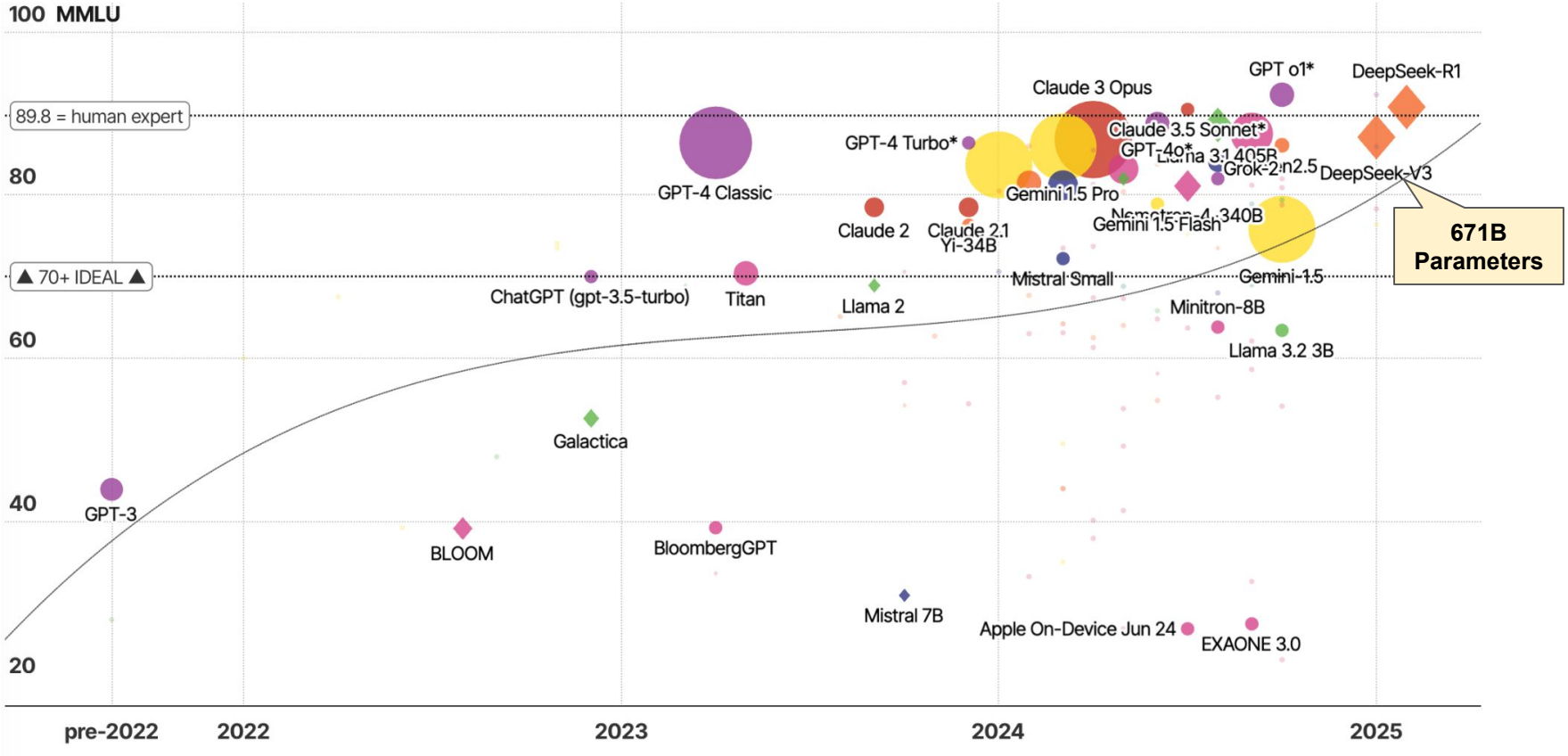
UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Scalable High-level Synthesis for AI Accelerator Design

Hanchen Ye, University of Illinois Urbana-Champaign
Advisor: Prof. Deming Chen
April 9, 2025

Final Exam Committee: Prof. Vikram Adve, Prof. Deming Chen (Chair), Prof. Jian Huang, Dr. Stephen Neuendorffer (Alphabetical Order)

Computation Demands of Deep Learning Grows Fast



Source: Information is Beautiful

Abstraction Gap between Application and Hardware

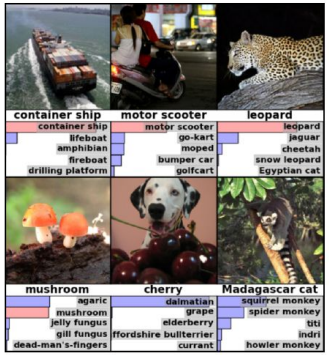
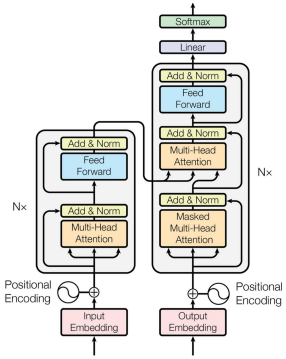
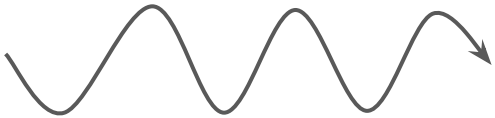


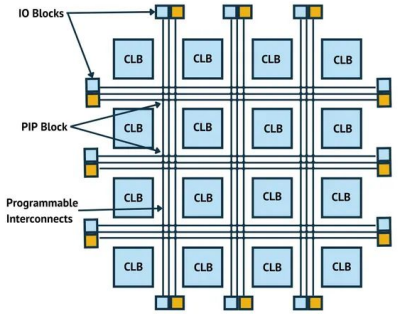
Image Classification



Transformers

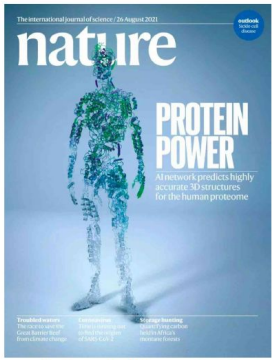


Huge Abstraction Gap

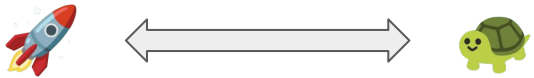


FPGA (Field-programmable Gate Arrays)

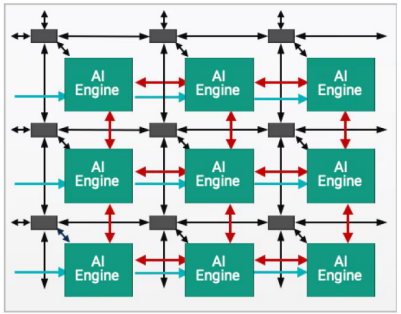
AlphaGo (Nature 2016)



AlphaFold (Nature 2021)



Huge Productivity Gap

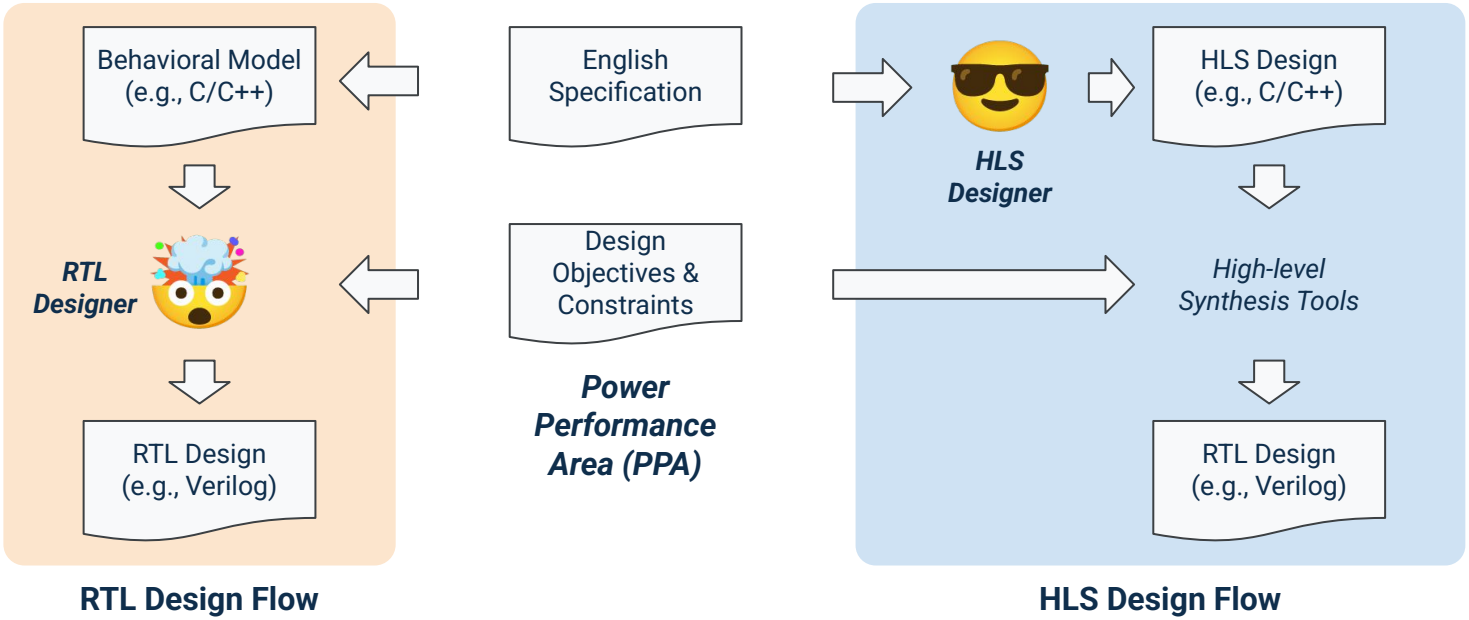


CGRA (Coarse Grain Reconfigurable Architecture)

Deep Learning Applications

Hardwares

High-level Synthesis (HLS)



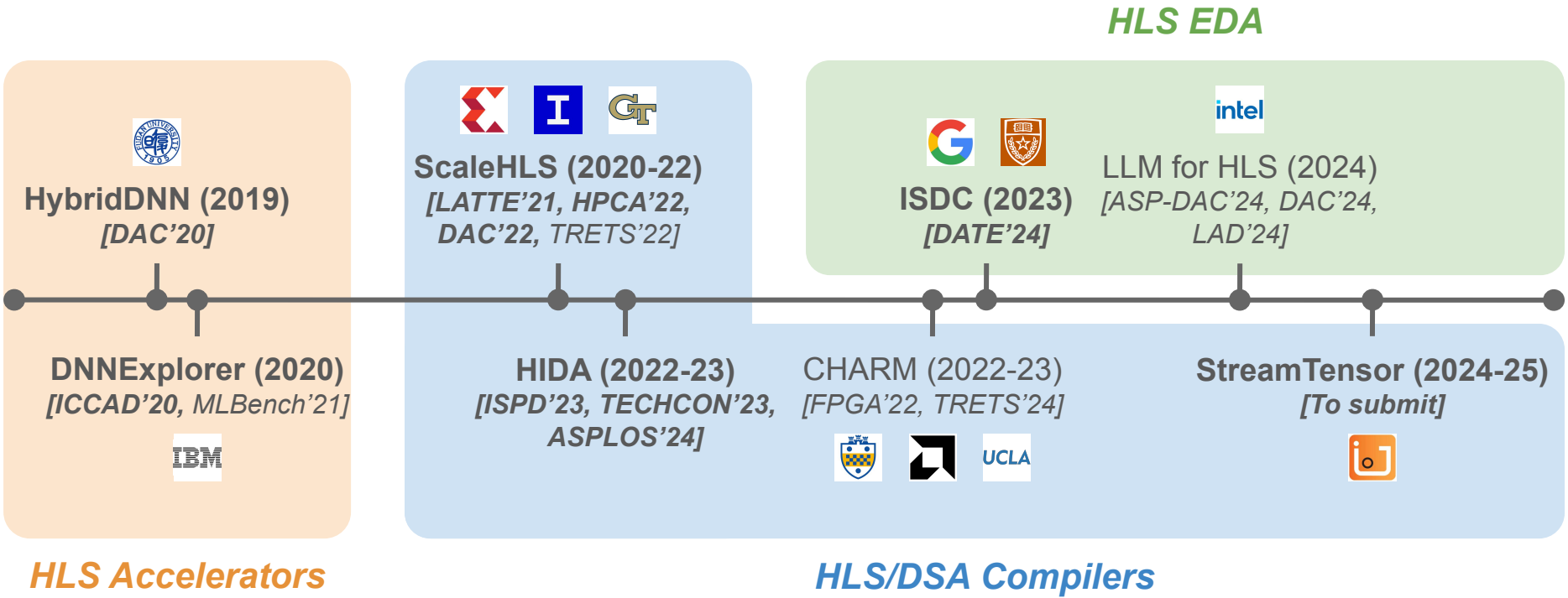
RTL Design Flow

HLS Design Flow

- **Manual** optimization and scheduling
- **Long** design cycle
- **Low** portability against different Process Design Kit (PDK) or PPA requirements

- **Automated** optimization and scheduling
- **Short** design cycle
- **High** portability against different PDK or PPA requirements

Research Overview



ScaleHLS: Single-kernel HLS Optimization

A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation

ScaleHLS Outline

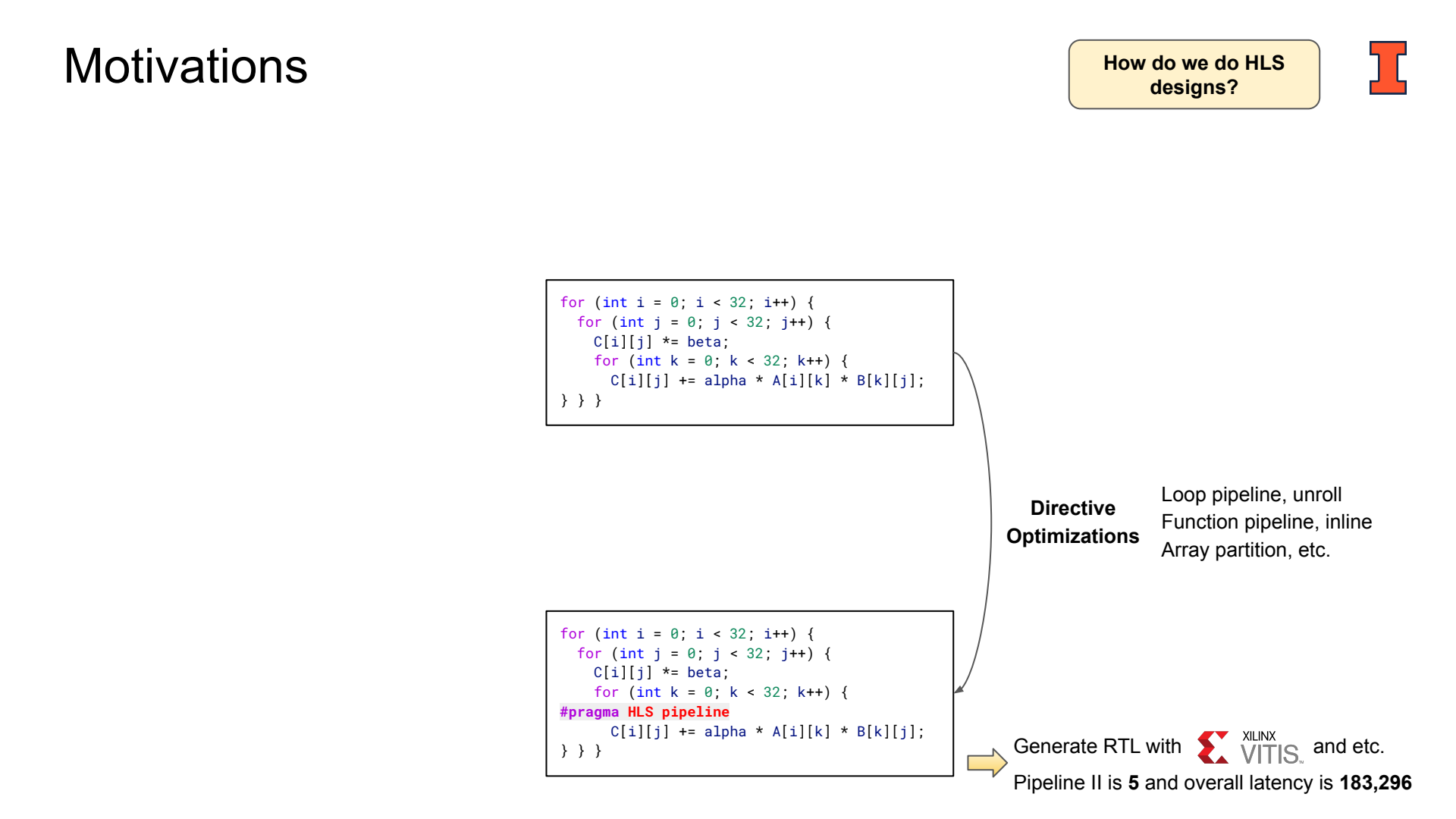
- Motivation
- ScaleHLS Optimizations
- ScaleHLS Design Space Exploration
- ScaleHLS Results

ScaleHLS Outline

- Motivation
- ScaleHLS Optimizations
- ScaleHLS Design Space Exploration
- ScaleHLS Results

Motivations

How do we do HLS designs?



```
for (int i = 0; i < 32; i++) {
  for (int j = 0; j < 32; j++) {
    C[i][j] *= beta;
    for (int k = 0; k < 32; k++) {
      C[i][j] += alpha * A[i][k] * B[k][j];
} } }
```

Directive Optimizations

Loop pipeline, unroll
Function pipeline, inline
Array partition, etc.

```
for (int i = 0; i < 32; i++) {
  for (int j = 0; j < 32; j++) {
    C[i][j] *= beta;
    for (int k = 0; k < 32; k++) {
#pragma HLS pipeline
      C[i][j] += alpha * A[i][k] * B[k][j];
} } }
```

Generate RTL with XILINX VITIS and etc.

Pipeline II is **5** and overall latency is **183,296**

Motivations (Cont.)

Difficulties:

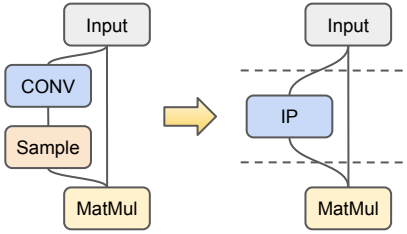
- Low-productive and error-proning
- Hard to enable automated design space exploration (DSE)
- NOT scalable! 💢

Solve problems at the 'correct' level AND automate it

MLIR

Approaches of *ScaleHLS*:

- Represent HLS designs at multiple levels of abstractions
- Make the *multi-level* optimizations automated and parameterized
- Enable an automated DSE
- End-to-end high-level analysis and optimization flow




```
for (int i = 0; i < 32; i++) {
  for (int j = 0; j < 32; j++) {
    C[i][j] *= beta;
    for (int k = 0; k < 32; k++) {
      C[i][j] += alpha * A[i][k] * B[k][j];
} } }
```

```
for (int k = 0; k < 32; k++) {
  for (int i = 0; i < 32; i++) {
    for (int j = 0; j < 32; j++) {
      if (k == 0)
        C[i][j] *= beta;
      C[i][j] += alpha * A[i][k] * B[k][j];
} } }
```

```
for (int k = 0; k < 32; k++) {
  for (int i = 0; i < 32; i++) {
    for (int j = 0; j < 32; j++) {
#pragma HLS pipeline
      if (k == 0)
        C[i][j] *= beta;
      C[i][j] += alpha * A[i][k] * B[k][j];
} } }
```

How do we do HLS designs?

Graph Optimizations
Node fusion
IP integration
Task-level pipeline, etc.

Manual Code Rewriting

Loop Optimizations
Loop interchange
Loop perfectization
Loop tile, skew, etc.

Manual Code Rewriting

Directive Optimizations
Loop pipeline, unroll
Function pipeline, inline
Array partition, etc.

Manual Code Rewriting

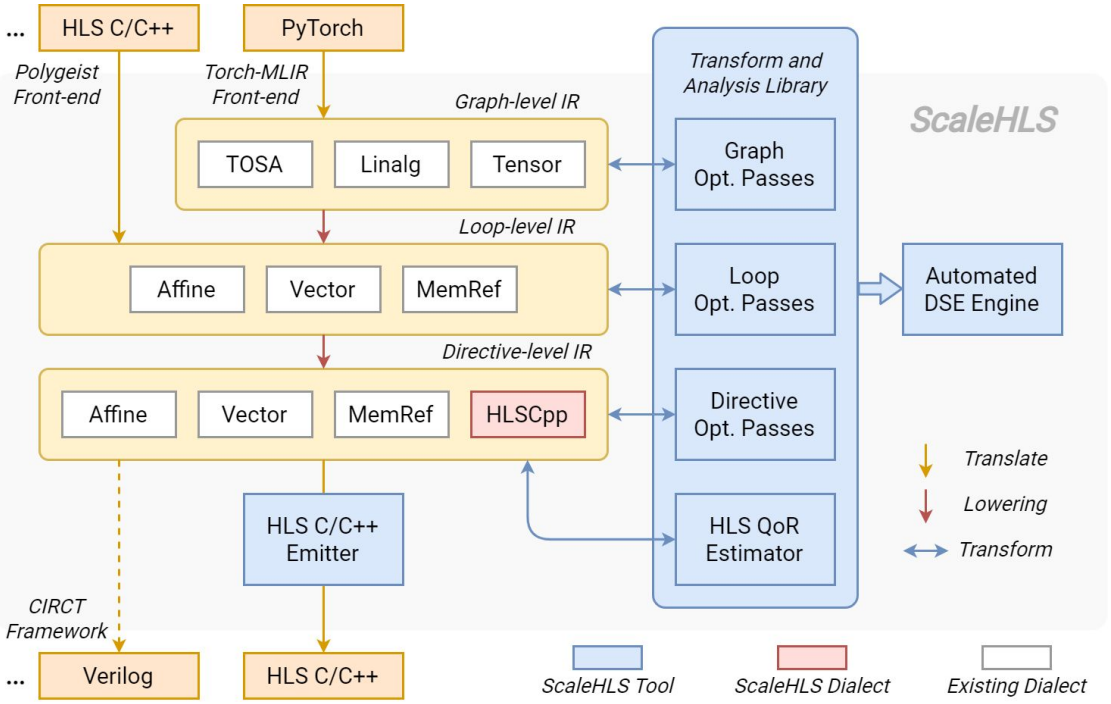
Generate RTL with XILINX VITIS and etc.

Pipeline II is **2** and overall latency is **65,552**

ScaleHLS Outline

- Motivation
- ScaleHLS Optimizations
- ScaleHLS Design Space Exploration
- ScaleHLS Results

ScaleHLS Framework



Represent It!

Graph-level IR: TOSA, Linalg, and Tensor dialect.

Loop-level IR: Affine and Memref dialect. Can leverage the MLIR built-in transforms and analyses.

Directive-level IR: HLSCpp, Affine, and Memref.

Optimize It!

Optimization Passes: Cover the graph, loop, and directive levels. Solve optimization problems at the 'correct' abstraction level. 💪

QoR Estimator: Estimate the latency and resource utilization through IR analysis.

Explore It!

Transform and Analysis Library: Parameterized interfaces of all optimization passes and the QoR estimator. A playground of DSE. 🚀

Automated DSE Engine: Find the Pareto-frontier of the throughput-area trade-off design space.

Enable End-to-end Flow!

HLS C/C++ Emitter: Generate synthesizable HLS designs for downstream tools, such as Vivado HLS.

Single-kernel Optimizations

	Passes	Target	Parameters
Graph	**-legalize-dataflow**	function	insert-copy
	-split-function	function	min-gran
Loop	**-affine-loop-perfectization**	loop band	-
	-affine-loop-order-opt	loop band	perm-map
	-remove-variable-bound	loop band	-
	-affine-loop-tile	loop	tile-size
	-affine-loop-unroll	loop	unroll-factor
Direct.	**-loop-pipelining**	loop	target-ii
	-func-pipelining	function	target-ii
	-array-partition	function	part-factors
Misc.	**-simplify-affine-if**	function	-
	-affine-store-forward	function	-
	-simplify-memref-access	function	-
	-canonicalize -cse	function	-

Boldface ones are new passes provided by us, while others are MLIR built-in passes.

```
void syrk(int alpha, int beta, int C[32][32], int A[32][32]) {
  for (int i = 0; i < 32; i++) {
    for (int j = 0; j <= i; j++) {
      C[i][j] *= beta;
      for (int k = 0; k < 32; k++) {
        C[i][j] += alpha * A[i][k] * A[j][k];
} } } }
```

Baseline C

Loop and Directive Opt in MLIR

```
void syrk(int alpha, int beta, int C[32][32], int A[32][32]) {
#pragma HLS interface s_axilite port=return bundle=ctrl
#pragma HLS interface s_axilite port=alpha bundle=ctrl
#pragma HLS interface s_axilite port=beta bundle=ctrl
#pragma HLS interface bram port=C
#pragma HLS interface bram port=A

#pragma HLS resource variable=C core=ram_s2p_bram

#pragma HLS array_partition variable=A cyclic factor=2 dim=2
#pragma HLS resource variable=A core=ram_s2p_bram

  for (int k = 0; k < 32; k += 2) {
    for (int i = 0; i < 32; i += 1) {
      for (int j = 0; j < 32; j += 1) {
#pragma HLS pipeline II = 3
        if ((i - j) >= 0) {
          int v7 = C[i][j];
          int v8 = beta * v7;
          int v9 = A[i][k];
          int v10 = A[j][k];
          int v11 = (k == 0) ? v8 : v7;
          int v12 = alpha * v9;
          int v13 = v12 * v10;
          int v14 = v11 + v13;
          int v15 = A[i][(k + 1)];
          int v16 = A[j][(k + 1)];
          int v17 = alpha * v15;
          int v18 = v17 * v16;
          int v19 = v14 + v18;
          C[i][j] = v19;
} } } } }
```

Optimized C emitted by the C/C++ emitter

Single-kernel Optimizations (Cont.)

Loop Order Permutation

- The minimum II (Initiation Interval) of a loop pipeline can be calculated as:

$$II_{min} = \max_{d} \left(\left\lceil \frac{Delay_d}{Distance_d} \right\rceil \right)$$

- $Delay_d$ and $Distance_d$ are the scheduling delay and distance (calculated from the dependency vector) of each loop-carried dependency d.

- To achieve a smaller II, the loop order permutation pass performs affine analysis and attempt to permute loops associated with loop-carried dependencies in order to maximize the $Distance$.

```
void syrk(int alpha, int beta, int C[32][32], int A[32][32]) {
  for (int i = 0; i < 32; i++) {
    for (int j = 0; j <= i; j++) {
      C[i][j] *= beta;
      for (int k = 0; k < 32; k++) {
        C[i][j] += alpha * A[i][k] * A[j][k];
} } } }
```

Loop perfectization

Baseline C

Loop and Directive Opt in MLIR

```
void syrk(int alpha, int beta, int C[32][32], int A[32][32]) {
#pragma HLS interface s_axilite port=return bundle=ctrl
#pragma HLS interface s_axilite port=alpha bundle=ctrl
#pragma HLS interface s_axilite port=beta bundle=ctrl
#pragma HLS interface bram port=C
#pragma HLS interface bram port=A

#pragma HLS resource variable=C core=ram_s2p_bram

#pragma HLS array_partition variable=A cyclic factor=2 dim=2
#pragma HLS resource variable=A core=ram_s2p_bram

  for (int k = 0; k < 32; k += 2) {
    for (int i = 0; i < 32; i += 1) {
      for (int j = 0; j < 32; j += 1) {
#pragma HLS pipeline II = 3
        if ((i - j) >= 0) {
          int v7 = C[i][j];
          int v8 = beta * v7;
          int v9 = A[i][k];
          int v10 = A[j][k];
          int v11 = (k == 0) ? v8 : v7;
          int v12 = alpha * v9;
          int v13 = v12 * v10;
          int v14 = v11 + v13;
          int v15 = A[i][(k + 1)];
          int v16 = A[j][(k + 1)];
          int v17 = alpha * v15;
          int v18 = v17 * v16;
          int v19 = v14 + v18;
          C[i][j] = v19;
} } } } }
```

Loop order permutation; Loop unroll

Remove variable loop bound

Optimized C emitted by the C/C++ emitter

Single-kernel Optimizations (Cont.)

Loop Pipelining

- Apply loop pipelining directives to a loop and set a targeted initiation interval.
- In the IR of ScaleHLS, directives are represented using the HLSCpp dialect. In the example, the pipelined %j loop is represented as:

```
affine.for %j = 0 to 32 {
  … …
} attributes {loop_directive = #hlscpp.ld<pipeline=1,
targetII=3, dataflow=0, flatten=0, … … >}
```

```
void syrk(int alpha, int beta, int C[32][32], int A[32][32]) {
  for (int i = 0; i < 32; i++) {
    for (int j = 0; j <= i; j++) {
      C[i][j] *= beta;
      for (int k = 0; k < 32; k++) {
        C[i][j] += alpha * A[i][k] * A[j][k];
} } } }
```

Loop perfectization

Baseline C

Loop and Directive Opt in MLIR

```
void syrk(int alpha, int beta, int C[32][32], int A[32][32]) {
#pragma HLS interface s_axilite port=return bundle=ctrl
#pragma HLS interface s_axilite port=alpha bundle=ctrl
#pragma HLS interface s_axilite port=beta bundle=ctrl
#pragma HLS interface bram port=C
#pragma HLS interface bram port=A

#pragma HLS resource variable=C core=ram_s2p_bram

#pragma HLS array_partition variable=A cyclic factor=2 dim=2
#pragma HLS resource variable=A core=ram_s2p_bram

  for (int k = 0; k < 32; k += 2) {
    for (int i = 0; i < 32; i += 1) {
      for (int j = 0; j < 32; j += 1) {
#pragma HLS pipeline II = 3
        if ((i - j) >= 0) {
          int v7 = C[i][j];
          int v8 = beta * v7;
          int v9 = A[i][k];
          int v10 = A[j][k];
          int v11 = (k == 0) ? v8 : v7;
          int v12 = alpha * v9;
          int v13 = v12 * v10;
          int v14 = v11 + v13;
          int v15 = A[i][(k + 1)];
          int v16 = A[j][(k + 1)];
          int v17 = alpha * v15;
          int v18 = v17 * v16;
          int v19 = v14 + v18;
          C[i][j] = v19;
} } } } }
```

Loop order permutation; Loop unroll

Remove variable loop bound

Loop pipeline

Optimized C emitted by the C/C++ emitter

Single-kernel Optimizations (Cont.)

Array Partition

- Array partition is one of the most important directives because the memories requires enough bandwidth to comply with the computation parallelism.
- The array partition pass analyzes the accessing pattern of each array and automatically select suitable partition fashion and factor.
- In the example, the %A array is accessed at address `[i,k]` and `[i,k+1]` simultaneously after pipelined, thus %A array is cyclically partitioned with two.

```
void syrk(int alpha, int beta, int C[32][32], int A[32][32]) {
  for (int i = 0; i < 32; i++) {
    for (int j = 0; j <= i; j++) {
      C[i][j] *= beta;
      for (int k = 0; k < 32; k++) {
        C[i][j] += alpha * A[i][k] * A[j][k];
} } } }
```

Loop perfectization

Baseline C

Loop and Directive Opt in MLIR

```
void syrk(int alpha, int beta, int C[32][32], int A[32][32]) {
#pragma HLS interface s_axilite port=return bundle=ctrl
#pragma HLS interface s_axilite port=alpha bundle=ctrl
#pragma HLS interface s_axilite port=beta bundle=ctrl
#pragma HLS interface bram port=C
#pragma HLS interface bram port=A

#pragma HLS resource variable=C core=ram_s2p_bram

#pragma HLS array_partition variable=A cyclic factor=2 dim=2
#pragma HLS resource variable=A core=ram_s2p_bram

  for (int k = 0; k < 32; k += 2) {
    for (int i = 0; i < 32; i += 1) {
      for (int j = 0; j < 32; j += 1) {
#pragma HLS pipeline II = 3
        if ((i - j) >= 0) {
          int v7 = C[i][j];
          int v8 = beta * v7;
          int v9 = A[i][k];
          int v10 = A[j][k];
          int v11 = (k == 0) ? v8 : v7;
          int v12 = alpha * v9;
          int v13 = v12 * v10;
          int v14 = v11 + v13;
          int v15 = A[i][(k + 1)];
          int v16 = A[j][(k + 1)];
          int v17 = alpha * v15;
          int v18 = v17 * v16;
          int v19 = v14 + v18;
          C[i][j] = v19;
} } } } }
```

Array partition

Loop order permutation; Loop unroll

Remove variable loop bound

Loop pipeline

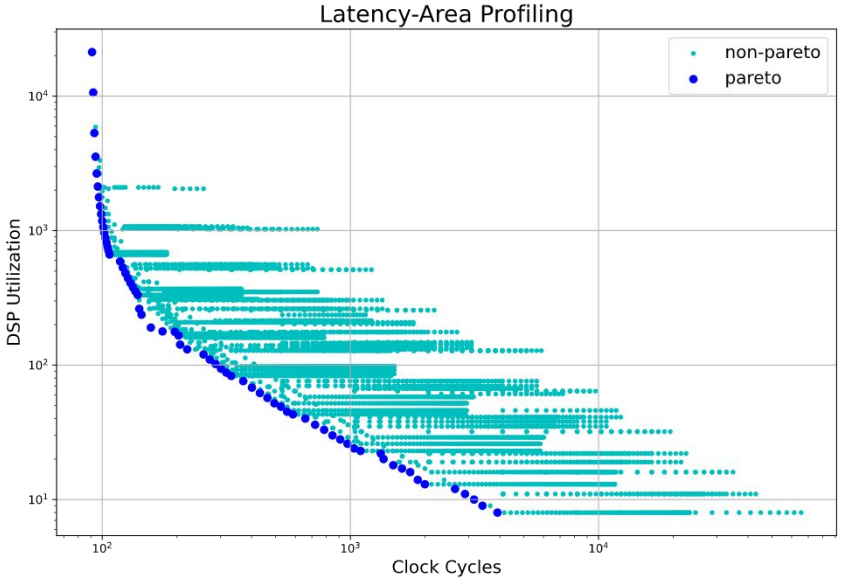
Simplify if ops;
Store ops forward;
Simplify memref ops

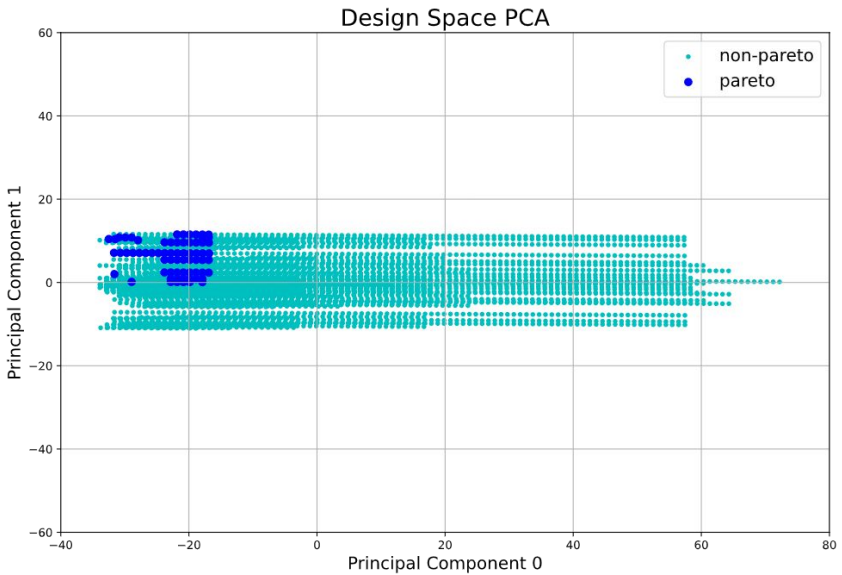
Optimized C emitted by the C/C++ emitter

ScaleHLS Outline

- Motivation
- ScaleHLS Optimizations
- ScaleHLS Design Space Exploration
- ScaleHLS Results

Single-kernel Design Space Exploration





Pareto frontier of a GEMM kernel

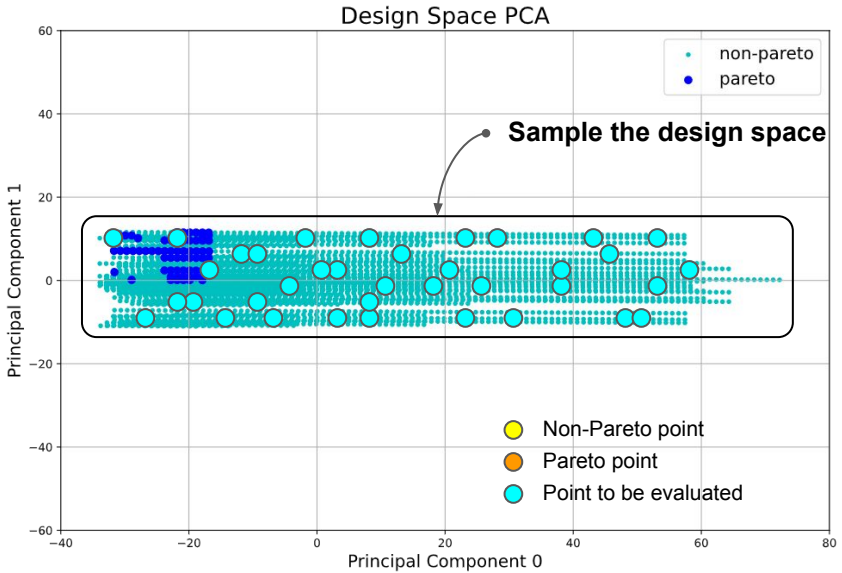
- Latency and area are profiled for each design point
- Dark blue points are Pareto points
- Loop perfectization, loop order permutation, loop tiling, loop pipelining, and array partition passes are involved

- Each parameter of a pass becomes one dimension, the original 4-dimensional design space is reduced to two dimensions through PCA
- Pareto points are located at a corner of the design space, the variance of Pareto points is much smaller than the overall variance

Single-kernel Design Space Exploration (Cont.)

DSE algorithm:

1. Sample the whole design space and evaluate each sampled design point with the QoR estimator

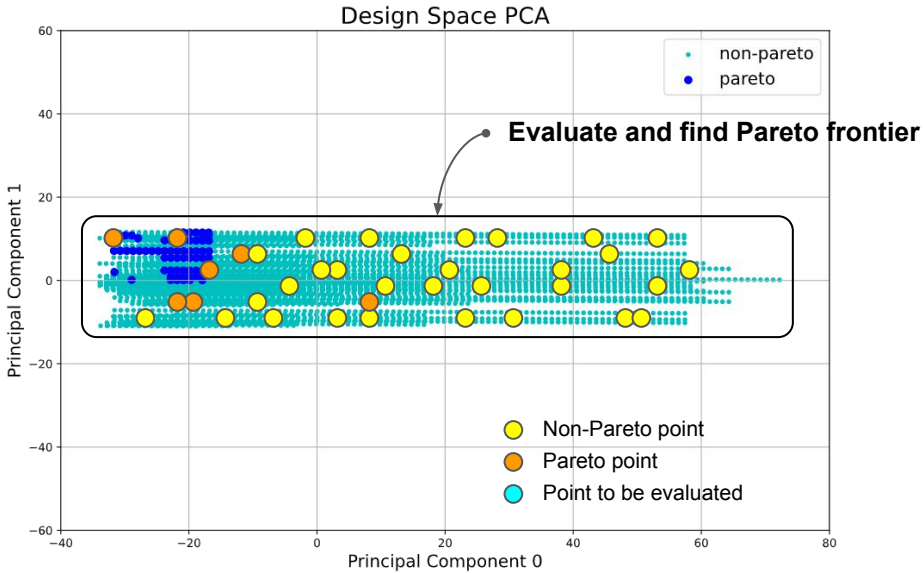


- Each parameter of a pass becomes one dimension, the original 4-dimensional design space is reduced to two dimensions through PCA
- Pareto points are located at a corner of the design space, the variance of Pareto points is much smaller than the overall variance

Single-kernel Design Space Exploration (Cont.)

DSE algorithm:

1. Sample the whole design space and evaluate each sampled design point with the QoR estimator
2. Extract the Pareto frontier from all evaluated design points

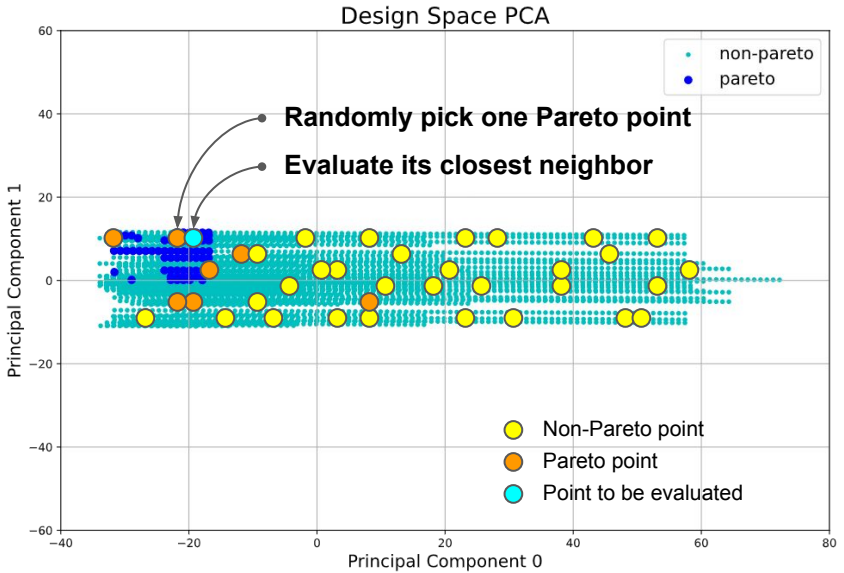


- Each parameter of a pass becomes one dimension, the original 4-dimensional design space is reduced to two dimensions through PCA
- Pareto points are located at a corner of the design space, the variance of Pareto points is much smaller than the overall variance

Single-kernel Design Space Exploration (Cont.)

DSE algorithm:

1. Sample the whole design space and evaluate each sampled design point with the QoR estimator
2. Extract the Pareto frontier from all evaluated design points
3. Evaluate the closest neighbor of a randomly selected design point in the current Pareto frontier

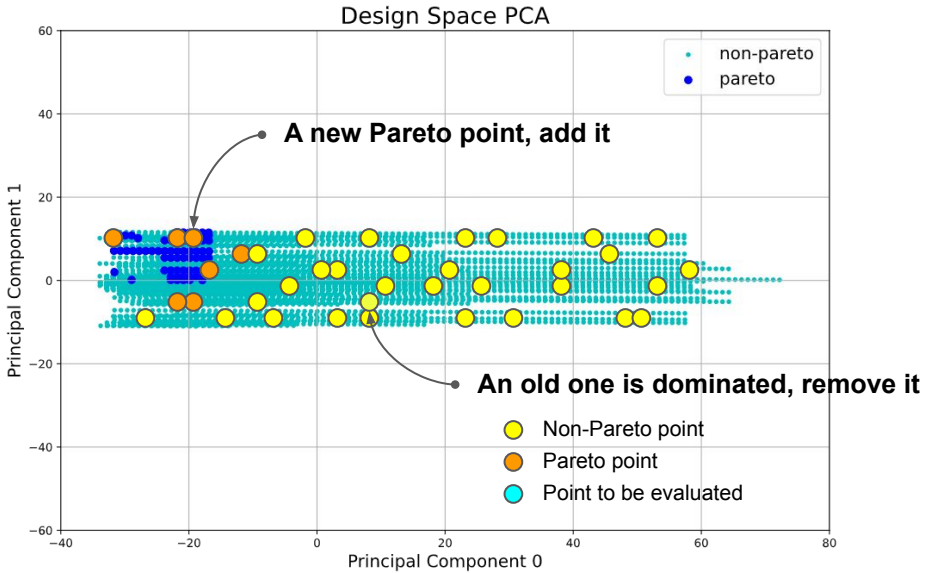


- Each parameter of a pass becomes one dimension, the original 4-dimensional design space is reduced to two dimensions through PCA
- Pareto points are located at a corner of the design space, the variance of Pareto points is much smaller than the overall variance

Single-kernel Design Space Exploration (Cont.)

DSE algorithm:

1. Sample the whole design space and evaluate each sampled design point with the QoR estimator
2. Extract the Pareto frontier from all evaluated design points
3. Evaluate the closest neighbor of a randomly selected design point in the current Pareto frontier
4. Repeat step (2) and (3) to update the discovered Pareto frontier



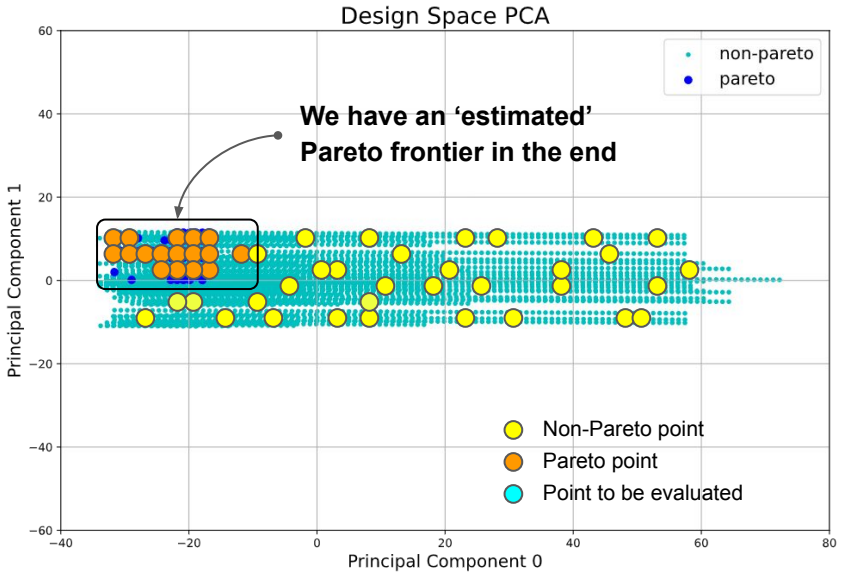
- Each parameter of a pass becomes one dimension, the original 4-dimensional design space is reduced to two dimensions through PCA
- Pareto points are located at a corner of the design space, the variance of Pareto points is much smaller than the overall variance

Single-kernel Design Space Exploration (Cont.)

DSE algorithm:

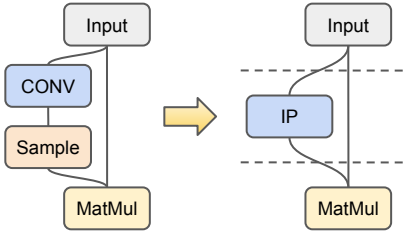
1. Sample the whole design space and evaluate each sampled design point with the QoR estimator
2. Extract the Pareto frontier from all evaluated design points
3. Evaluate the closest neighbor of a randomly selected design point in the current Pareto frontier
4. Repeat step (2) and (3) to update the discovered Pareto frontier
5. Stop when no eligible neighbor can be found or meeting the early-termination criteria

Given the **Transform and Analysis Library** provided by ScaleHLS, the DSE engine can be extended to support other optimization algorithms in the future.

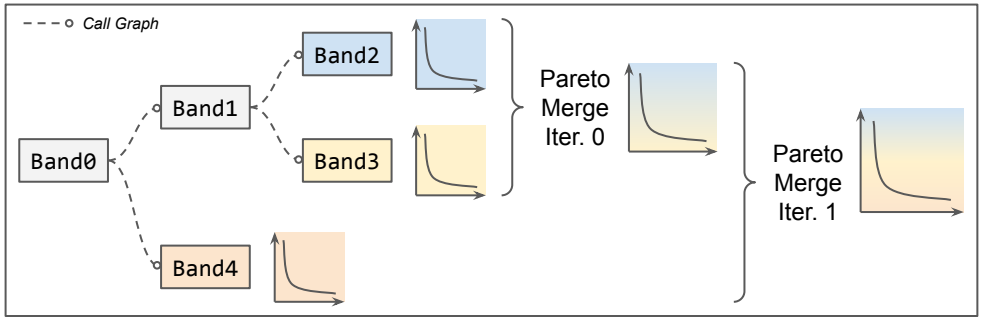


- Each parameter of a pass becomes one dimension, the original 4-dimensional design space is reduced to two dimensions through PCA
- Pareto points are located at a corner of the design space, the variance of Pareto points is much smaller than the overall variance

Multi-kernel Design Space Exploration



Graph Optimizations



Step (2) Global multi-kernel Pareto curving merging

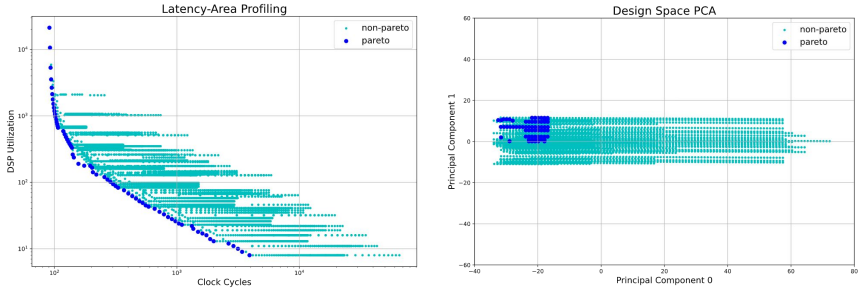
Loop Optimizations

Directive Optimizations

```
for (int i = 0; i < 32; i++) {
  for (int j = 0; j < 32; j++) {
    C[i][j] *= beta;
    for (int k = 0; k < 32; k++) {
      C[i][j] += alpha * A[i][k] * B[k][j];
} } }
```

```
for (int k = 0; k < 32; k++) {
  for (int i = 0; i < 32; i++) {
    for (int j = 0; j < 32; j++) {
      if (k == 0)
        C[i][j] *= beta;
      C[i][j] += alpha * A[i][k] * B[k][j];
} } }
```

```
for (int k = 0; k < 32; k++) {
  for (int i = 0; i < 32; i++) {
    for (int j = 0; j < 32; j++) {
#pragma HLS pipeline
      if (k == 0)
        C[i][j] *= beta;
      C[i][j] += alpha * A[i][k] * B[k][j];
} } }
```



Step (1) Local single-kernel *loop* and *directive* DSE

ScaleHLS Outline

- Motivation
- ScaleHLS Optimizations
- ScaleHLS Design Space Exploration
- ScaleHLS Results

Experimental Results of C/C++ Kernel

Kernel	Prob. Size	Speedup	LP	RVB	Perm. Map	Tiling Sizes	Pipeline II	Array Partition
BICG	4096	41.7×	No	No	[1, 0]	[16, 8]	43	A:[8, 16], s:[16], q:[8], p:[16], r:[8]
GEMM	4096	768.1×	Yes	No	[1, 2, 0]	[8, 1, 16]	3	C:[1, 16], A:[1, 8], B:[8, 16]
GESUMMV	4096	199.1×	Yes	No	[1, 0]	[8, 16]	9	A:[16, 8], B:[16, 8], tmp:[16], x:[8], y:[16]
SYR2K	4096	384.0×	Yes	Yes	[1, 2, 0]	[8, 4, 4]	8	C:[4, 4], A:[4, 8], B:[4, 8]
SYRK	4096	384.1×	Yes	Yes	[1, 2, 0]	[64, 1, 1]	3	C:[1, 1], A:[1, 64]
TRMM	4096	590.9×	Yes	Yes	[1, 2, 0]	[4, 4, 32]	13	A:[4, 4], B:[4, 32]

1. The target platform is Xilinx XC7Z020 FPGA, which is an edge FPGA with 4.9 Mb memories, 220 DSPs, and 53,200 LUTs. The data types of all kernels are single-precision floating-points.
2. Among all six benchmarks, a **speedup** ranging from 41.7× to 768.1× is obtained compared to the baseline design, which is the original computation kernel from PolyBench-C without the optimization of DSE.
3. **LP** and **RVB** denote Loop Perfectization and Remove Variable Bound, respectively.
4. In the Loop Order Optimization (**Perm. Map**), the i-th loop in the loop nest is permuted to location $PermMap$ [i], where locations are from the outermost loop to inner.

Experimental Results of DNN Models

Model	Speedup	Runtime (seconds)	Memory (SLR Util. %)	DSP (SLR Util. %)	LUT (SLR Util. %)	FF (SLR Util. %)	Our DSP Effi. (OPs/Cycle/DSP)	DSP Effi. of TVM-VTA [26]
ResNet-18	3825.0×	60.8	91.7Mb (79.5%)	1326 (58.2%)	157902 (40.1%)	54766 (6.9%)	1.343	0.344
VGG-16	1505.3×	37.3	46.7Mb (40.5%)	878 (38.5%)	88108 (22.4%)	31358 (4.0%)	0.744	0.296
MobileNet	1509.0×	38.1	79.4Mb (68.9%)	1774 (77.8%)	138060 (35.0%)	56680 (7.2%)	0.791	0.468

1. The target platform is one SLR (super logic region) of Xilinx VU9P FPGA which is a large FPGA containing 115.3 Mb memories, 2280 DSPs and 394,080 LUTs on each SLR.
2. The PyTorch implementations are parsed into ScaleHLS and optimized using the proposed multi-level optimization methodology.
3. By combining the graph, loop, and directive levels of optimization, a **speedup** ranging from 1505.3× to 3825.0× is obtained compared to the baseline designs, which are compiled from PyTorch to HLS C/C++ through ScaleHLS but without the multi-level optimization applied.

HIDA: Multi-kernel HLS Optimization

A Hierarchical Dataflow Compiler for High-Level Synthesis

HIDA Outline

- Motivation
- HIDA Design Space Exploration
- HIDA Results

HIDA Outline

- Motivation
- HIDA Design Space Exploration
- HIDA Results

Motivation - Limitation of ScaleHLS

```
1  float A[32][16];
2  NODE0_I: for (int i=0; i<32; i++)
3    NODE0_K: for (int k=0; k<16; k++)
4      A[i][k] = ...; // Load array A.
5
6  float B[16][16];
7  NODE1_K: for (int k=0; k<16; k++)
8    NODE1_J: for (int j=0; j<16; j++)
9      B[k][j] = ...; // Load array B.
10
11 float C[16][16];
12 NODE2_I: for (int i=0; i<16; i++)
13   NODE2_J: for (int j=0; j<16; j++)
14     NODE2_K: for (int k=0; k<16; k++)
15       C[i][j] = A[i*2][k] * B[k][j];
```

Inter-kernel Correlation

- Node0 is connected to Node2 through buffer A
  - If buffer A is on-chip, the partition strategy of A is HIGHLY correlated with the parallel strategies of both Node0 and Node2
- Node1 is connected to Node2 through buffer B
  - Same as above

Connectedness

- Node0, 1, and 2 have different trip count: 32*16, 16*16, and 16*16*16
  - To enable efficient pipeline execution of Node0, 1, and 2, their latencies after parallelization should be similar

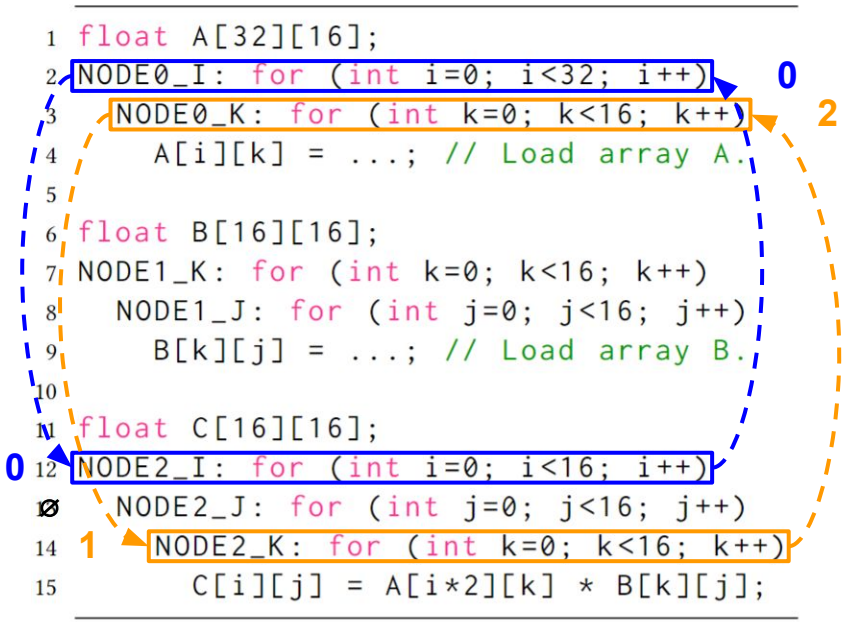
Intensity

Simply merging the local Pareto curves will not work well!

HIDA Outline

- Motivation
- HIDA Design Space Exploration
- HIDA Results

Multi-kernel Design Space Exploration

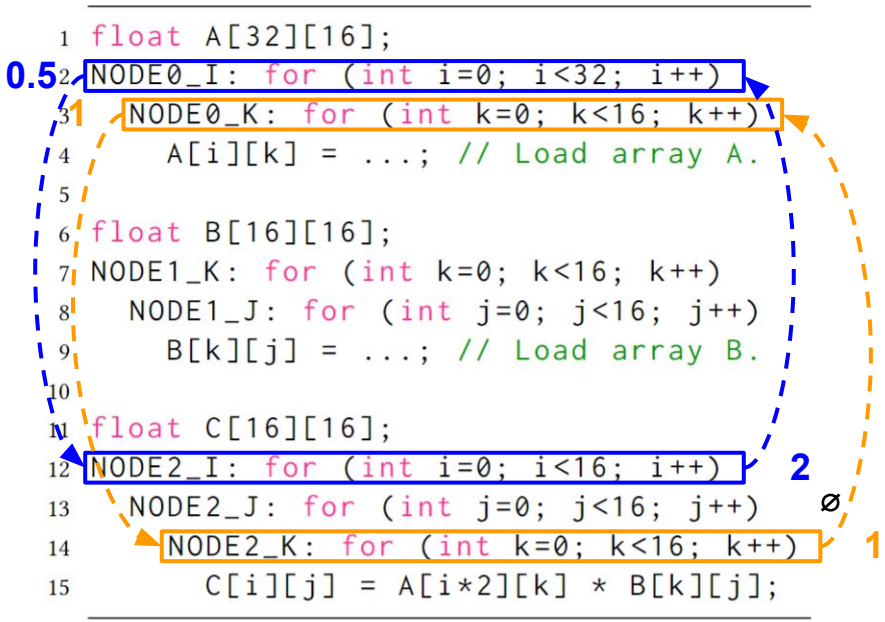


Step (1) Connectedness Analysis

Source	Target	Buffer	Permutation Map		Scaling Map	
			S-to-T	T-to-S	S-to-T	T-to-S
Node0	Node2	A	[0, ∅, 1]	[0, 2]	[0.5, 1]	[2, ∅, 1]
Node1	Node2	B	[∅, 1, 0]	[2, 1]	[1, 1]	[∅, 1, 1]

- **Permutation Map**
  - Record the alignment between loops

Multi-kernel Design Space Exploration (Cont.)



Step (1) Connectedness Analysis

Source	Target	Buffer	Permutation Map		Scaling Map	
			S-to-T	T-to-S	S-to-T	T-to-S
Node0	Node2	A	[0, ∅, 1]	[0, 2]	[0.5, 1]	[2, ∅, 1]
Node1	Node2	B	[∅, 1, 0]	[2, 1]	[1, 1]	[∅, 1, 1]

- **Permutation Map**
  - Record the alignment between loops
- **Scaling Map**
  - Record the alignment between strides
- **Affine Analysis-based**
  - Demand preprocessing: Loop normalize and perfectize, memory canonicalize

Multi-kernel Design Space Exploration (Cont.)

```
1  float A[32][16];
2  NODE0_I: for (int i=0; i<32; i++)
3    NODE0_K: for (int k=0; k<16; k++)
4      A[i][k] = ...; // Load array A.
5
6  float B[16][16];
7  NODE1_K: for (int k=0; k<16; k++)
8    NODE1_J: for (int j=0; j<16; j++)
9      B[k][j] = ...; // Load array B.
10
11 float C[16][16];
12 NODE2_I: for (int i=0; i<16; i++)
13   NODE2_J: for (int j=0; j<16; j++)
14     NODE2_K: for (int k=0; k<16; k++)
15       C[i][j] = A[i*2][k] * B[k][j];
```

Step (2) Node Sorting

Node	Connectedness	Intensity
Node0	1	512
Node1	1	256
Node2	2	4096

- **Descending Order of Connectedness**
  - Higher-connectedness node will affect more nodes
- **Intensity as Tie-breaker**
  - Higher-intensity nodes are more computationally complex, being more sensitive to optimization
- **Order: Node2 -> Node0 -> Node1**

Multi-kernel Design Space Exploration (Cont.)

```
1  float A[32][16];
2  NODE0_I: for (int i=0; i<32; i++)
3    NODE0_K: for (int k=0; k<16; k++)
4      A[i][k] = ...; // Load array A.
5
6  float B[16][16];
7  NODE1_K: for (int k=0; k<16; k++)
8    NODE1_J: for (int j=0; j<16; j++)
9      B[k][j] = ...; // Load array B.
10
11 float C[16][16];
12 NODE2_I: for (int i=0; i<16; i++)
13   NODE2_J: for (int j=0; j<16; j++)
14     NODE2_K: for (int k=0; k<16; k++)
15       C[i][j] = A[i*2][k] * B[k][j];
```

Step (3) Node Parallelization

- **Assuming maximum parallel factor is 32**
- **Node2 Parallelization: [4, 8, 1]**
  - Overall parallel factor is 32
  - ScaleHLS DSE without constraints
  - Solution unroll factors: [4, 8, 1]

Multi-kernel Design Space Exploration (Cont.)

```
1  float A[32][16];
2  NODE0_I: for (int i=0; i<32; i++)
3    NODE0_K: for (int k=0; k<16; k++)
4      A[i][k] = ...; // Load array A.
5
6  float B[16][16];
7  NODE1_K: for (int k=0; k<16; k++)
8    NODE1_J: for (int j=0; j<16; j++)
9      B[k][j] = ...; // Load array B.
10
11 float C[16][16];
12 NODE2_I: for (int i=0; i<16; i++)
13   NODE2_J: for (int j=0; j<16; j++)
14     NODE2_K: for (int k=0; k<16; k++)
15       C[i][j] = A[i*2][k] * B[k][j];
```

Source	Target	Buffer	Permutation Map		Scaling Map	
			S-to-T	T-to-S	S-to-T	T-to-S
Node0	Node2	A	[0, ∅, 1]	[0, 2]	[0.5, 1]	[2, ∅, 1]
Node1	Node2	B	[∅, 1, 0]	[2, 1]	[1, 1]	[∅, 1, 1]

Step (3) Node Parallelization

- **Assuming maximum parallel factor is 32**
- **Node2 Parallelization: [4, 8, 1]**
- **Node0 Parallelization: [4, 1]**
  - Overall parallel factor is 4, calculated from intensities of Node0 and 2 (32*512/4096)
  - ScaleHLS DSE with connectedness constraints, the unroll factors must NOT be mutually indivisible with constraints
    - Multiply with scaling map:
    - [4, 8, 1] ⊙ [2, ∅, 1] = [8, ∅, 1]
    - Permute with permutation map:
    - permute([8, ∅, 1], [0, 2]) = [8, 1]
  - Solution unroll factors: [4, 1]

Multi-kernel Design Space Exploration (Cont.)

```
1  float A[32][16];
2  NODE0_I: for (int i=0; i<32; i++)
3    NODE0_K: for (int k=0; k<16; k++)
4      A[i][k] = ...; // Load array A.
5
6  float B[16][16];
7  NODE1_K: for (int k=0; k<16; k++)
8    NODE1_J: for (int j=0; j<16; j++)
9      B[k][j] = ...; // Load array B.
10
11 float C[16][16];
12 NODE2_I: for (int i=0; i<16; i++)
13   NODE2_J: for (int j=0; j<16; j++)
14     NODE2_K: for (int k=0; k<16; k++)
15       C[i][j] = A[i*2][k] * B[k][j];
```

Source	Target	Buffer	Permutation Map		Scaling Map	
			S-to-T	T-to-S	S-to-T	T-to-S
Node0	Node2	A	[0, ∅, 1]	[0, 2]	[0.5, 1]	[2, ∅, 1]
Node1	Node2	B	[∅, 1, 0]	[2, 1]	[1, 1]	[∅, 1, 1]

Step (3) Node Parallelization

- **Assuming maximum parallel factor is 32**
- **Node2 Parallelization: [4, 8, 1]**
- **Node0 Parallelization: [4, 1]**
- **Node1 Parallelization: [1, 2]**
  - Overall parallel factor is 2, calculated from intensities of Node0 and 1 (32*256/4096)
  - ScaleHLS DSE with connectedness constraints
  - Solution unroll factors: [1, 2]

Multi-kernel Design Space Exploration (Cont.)

```
1  float A[32][16];
2  NODE0_I: for (int i=0; i<32; i++)
3    NODE0_K: for (int k=0; k<16; k++)
4      A[i][k] = ...; // Load array A.
5
6  float B[16][16];
7  NODE1_K: for (int k=0; k<16; k++)
8    NODE1_J: for (int j=0; j<16; j++)
9      B[k][j] = ...; // Load array B.
10
11 float C[16][16];
12 NODE2_I: for (int i=0; i<16; i++)
13   NODE2_J: for (int j=0; j<16; j++)
14     NODE2_K: for (int k=0; k<16; k++)
15       C[i][j] = A[i*2][k] * B[k][j];
```

Step (3) Node Parallelization

Node	Intensity	Parallel Factor		Loop Unroll Factors			
		w/o IA	w/ IA	IA+CA	IA	CA	Naive
Node0	512	32	4	[4, 1]	[2, 2]	[8, 4]	[4, 8]
Node1	256	32	2	[1, 2]	[1, 2]	[4, 8]	[4, 8]
Node2	4,096	32	32	[4, 8, 1]	[4, 8, 1]	[4, 8, 1]	[4, 8, 1]

Intensity-aware (IA) Connectedness-aware (CA) HIDA DSE

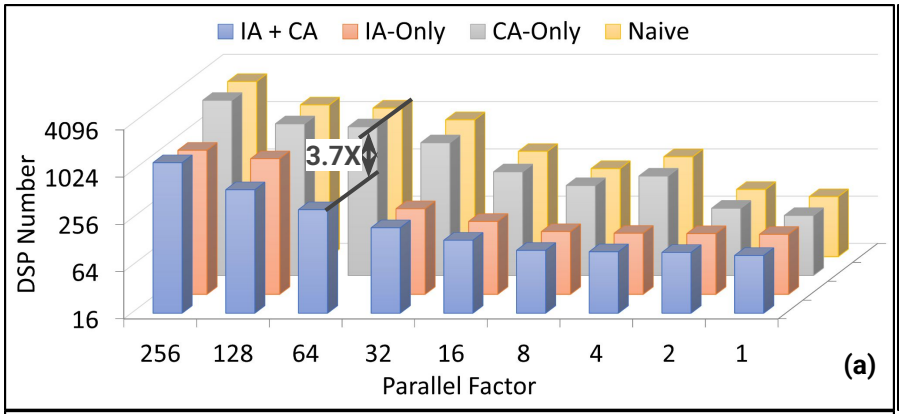
Naive ScaleHLS DSE

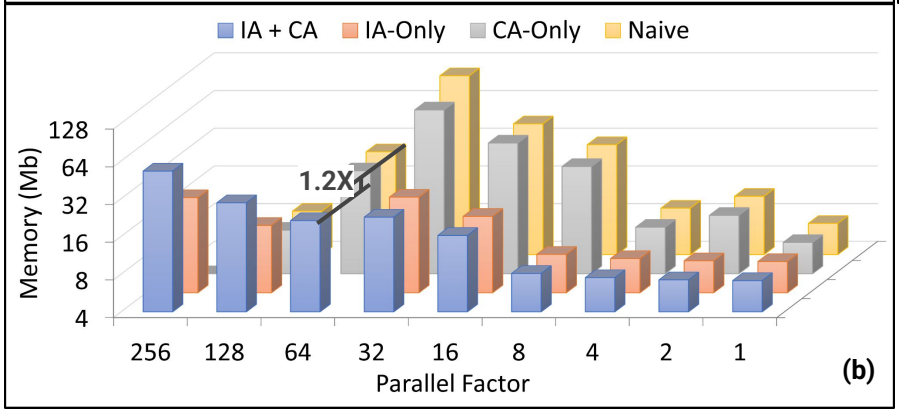
Array	Array Partition Factors				Bank Number				
	IA+CA	IA	CA	Naive	IA+CA	IA	CA	Naive	
A	[8, 1]	[8, 2]	[8, 4]	[8, 8]	8	16	32	64	***8x***
B	[1, 8]	[2, 8]	[4, 8]	[8, 8]	8	16	32	64	***8x***
C	[4, 8]	[4, 8]	[4, 8]	[4, 8]	32	32	32	32	***1x***

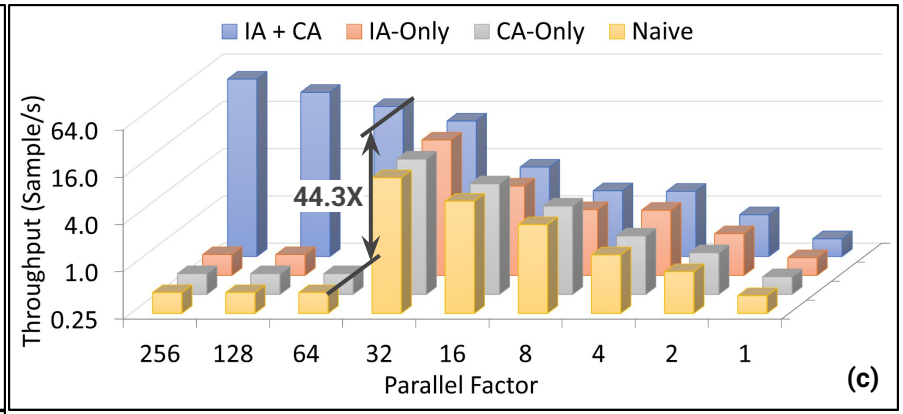
HIDA Outline

- Motivation
- HIDA Design Space Exploration
- HIDA Results

ResNet-18 Ablation Study

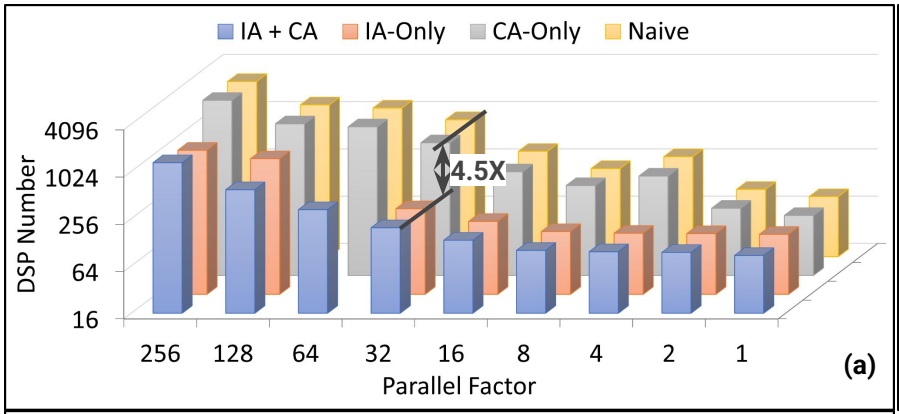


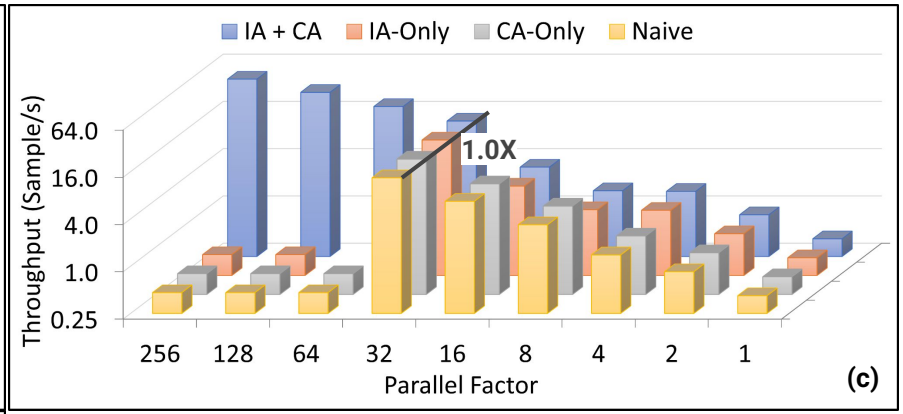


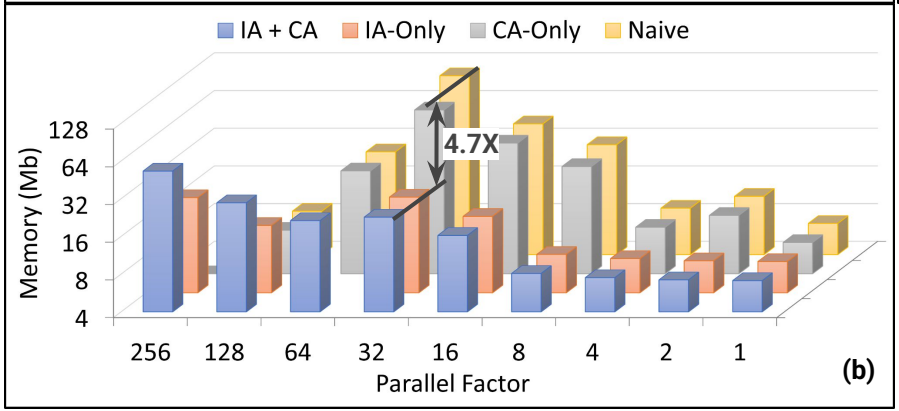


- IA+CA parallelization can determine whether the solution is scalable

ResNet-18 Ablation Study (Cont'd)







- IA+CA parallelization can determine whether the solution is scalable
- IA+CA parallelization can significantly reduce resource utilization

Experimental Results of C/C++ Kernel

Kernel	HIDA Compile Time (s)	LUT Number	FF Number	DSP Number	Throughput (Samples/s)*			
					HIDA	ScaleHLS [68]	SOFF [37]	Vitis [34]
2mm	0.65	38.8k	27.4k	269	239.22	122.39 (1.95×)	30.67 (7.80×)	1.23 (194.88×)
3mm	0.79	38.7k	27.8k	243	175.43	92.33 (1.90×)	-	1.04 (167.99×)
atax	2.06	44.6k	34.6k	260	1,021.39	932.26 (1.10×)	2,173.17 (0.47×)	103.18 (9.90×)
bicg	0.72	16.0k	15.1k	61	2,869.69	2,869.61 (1.00×)	2,295.75 (1.25×)	104.19 (27.54×)
correlation	0.91	14.5k	12.3k	66	67.33	59.77 (1.13×)	3.96 (16.99×)	1.32 (50.97×)
gesummv	0.60	34.2k	22.8k	232	31,685.68	31,685.68 (1.00×)	3,466.70 (9.14×)	266.65 (118.83×)
jacobi-2d	1.98	91.4k	56.6k	352	257.27	128.63 (2.00×)	-	2.71 (94.95×)
mvt	0.42	23.8k	16.5k	162	9,979.04	4,989.02 (2.00×)	870.01 (11.47×)	62.13 (160.62×)
seidel-2d	3.59	5.5k	2.5k	4	0.14	0.14 (1.00×)	-	0.11 (1.28×)
symm	1.05	14.9k	9.5k	74	2.62	2.62 (1.00×)	-	2.02 (1.29×)
syr2k	0.69	14.3k	12.8k	78	27.68	27.67 (1.00×)	-	1.44 (19.23×)
Geo. Mean	**0.99**					**1.29×**	**4.49×**	**31.08×**

* Numbers in () show throughput improvements of HIDA over others.

Experimental Results of DNN Models

Model	HIDA Compile Time (s)	LUT Number	DSP Number	Throughput (Samples/s)*			DSP Efficiency*		
				HIDA	DNNBuilder [75]	ScaleHLS [68]	HIDA	DNNBuilder [75]	ScaleHLS [68]
ResNet-18	83.1	142.1k	667	45.4	-	3.3 (13.88×)	73.8%	-	5.2% (14.24×)
MobileNet	110.8	132.9k	518	137.4	-	15.4 (8.90×)	75.5%	-	9.6% (7.88×)
ZFNet	116.2	103.8k	639	90.4	112.2 (0.81×)	-	82.8%	79.7% (1.04×)	-
VGG-16	199.9	266.2k	1118	48.3	27.7 (1.74×)	6.9 (6.99×)	102.1%	96.2% (1.06×)	18.6% (5.49×)
YOLO	188.2	202.8k	904	33.7	22.1 (1.52×)	-	94.3%	86.0% (1.10×)	-
MLP	40.9	21.0k	164	938.9	-	152.6 (6.15×)	90.0%	-	17.6% (5.10×)
Geo. Mean	**108.7**				**1.29×**	**8.54×**		**1.07×**	**7.49×**

* Numbers in () show throughput/DSP efficiency improvements of HIDA over others.

StreamTensor: Stream-based HLS Optimization

Make Tensors Stream in Dataflow Architectures

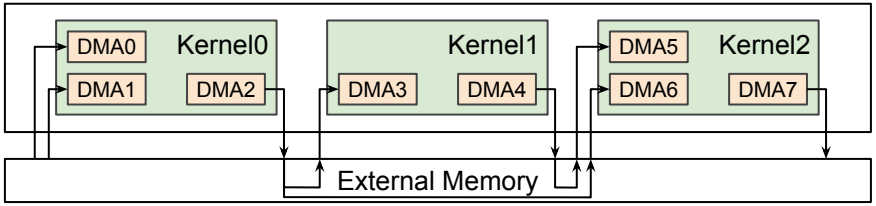
StreamTensor Outline

- Motivation
- StreamTensor Typing System
- StreamTensor Compilation Pipeline
- StreamTensor Design Spaces
- StreamTensor Results

StreamTensor Outline

- Motivation
- StreamTensor Typing System
- StreamTensor Compilation Pipeline
- StreamTensor Design Spaces
- StreamTensor Results

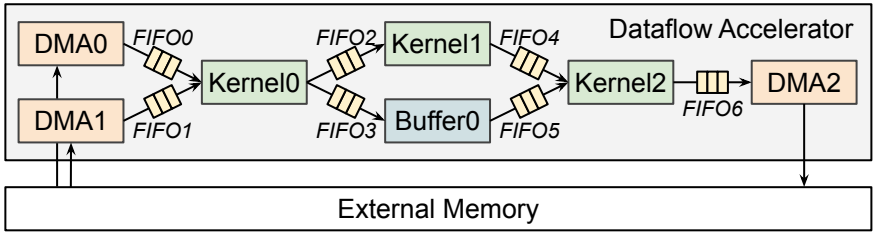
Motivation - Limitation of HIDA



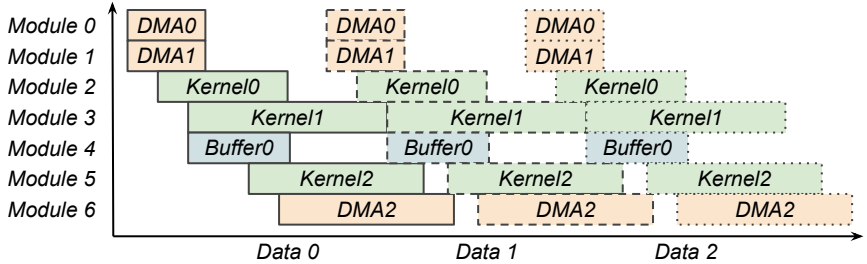
(a) A buffer-based dataflow accelerator

- Intermediate results are communicated through **on-chip ping-pong buffers**
- If on-chip memory resources are not enough, external memory is used
- Lead to frequent external memory access

Stream-based Kernel Fusion



(b) A stream-based dataflow accelerator



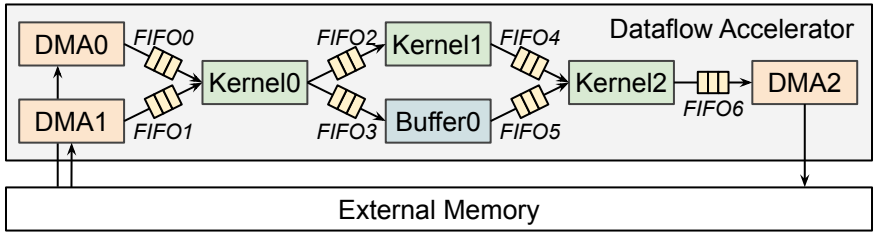
(c) Schedule of the stream-based dataflow accelerator

Reduce on-chip buffer utilization ✅

Reduce external memory access ✅

Reduce overall latency & throughput ✅

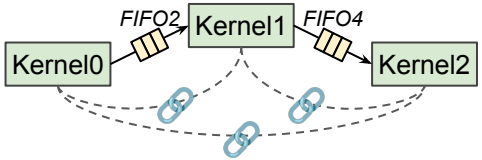
Pitfalls of Stream-based Dataflow Accelerator



Pitfall 1 ❎ External Memory Access

- Best external memory layout matching stream pattern?
- Widen external memory interfaces to maximize bandwidth?

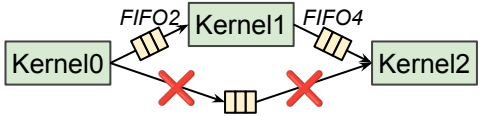
Pitfall 2 ❎
Inter-kernel Correlation



Kernel Correlated

- Kernel tiling & parallelization?
- Local buffer partition?
- Kernel loop permutation?

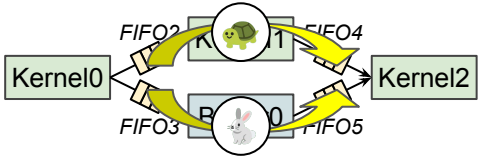
Pitfall 3 ❎
Kernel Fusion



Stream Layout/Order Mismatch

- Possible to stream?
- Minimal stream buffer size?
- Global fusion strategy?

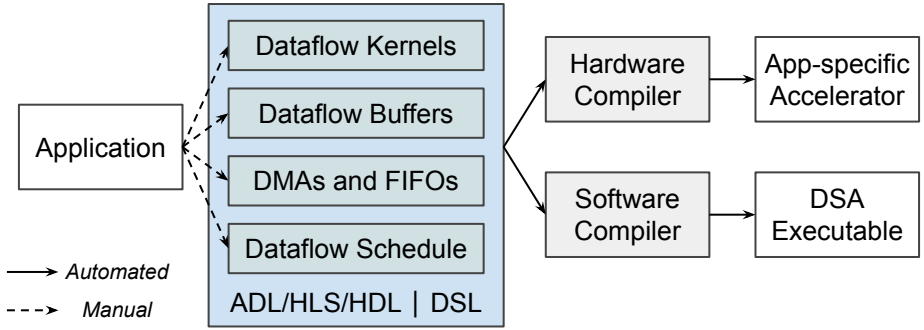
Pitfall 4 ❎
FIFO Sizing



Stream Deadlock

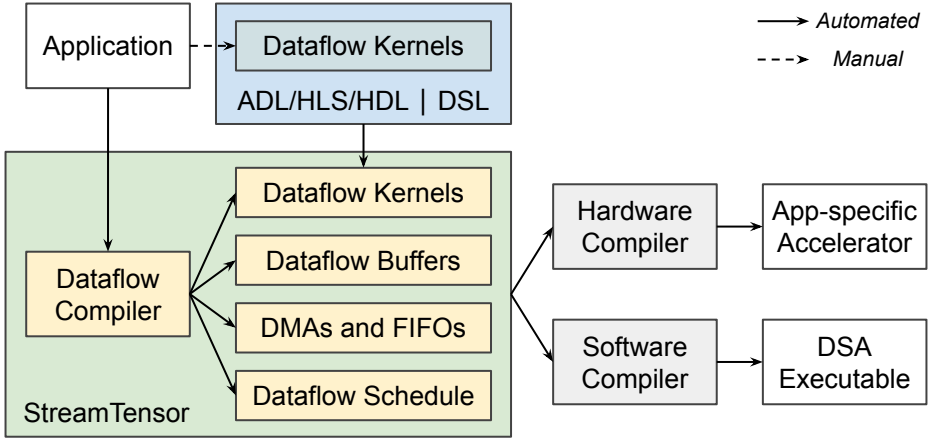
- Latency mismatch on different paths between two kernels
- Minimal FIFO size to avoid deadlock or kernel stall?

Dataflow Accelerator Design Paradigm



- Manual dataflow kernels, buffers, DMAs, and FIFOs design
- Difficult to comprehend the optimal solutions of the pitfalls
- Low design productivity

Paradigm Shift

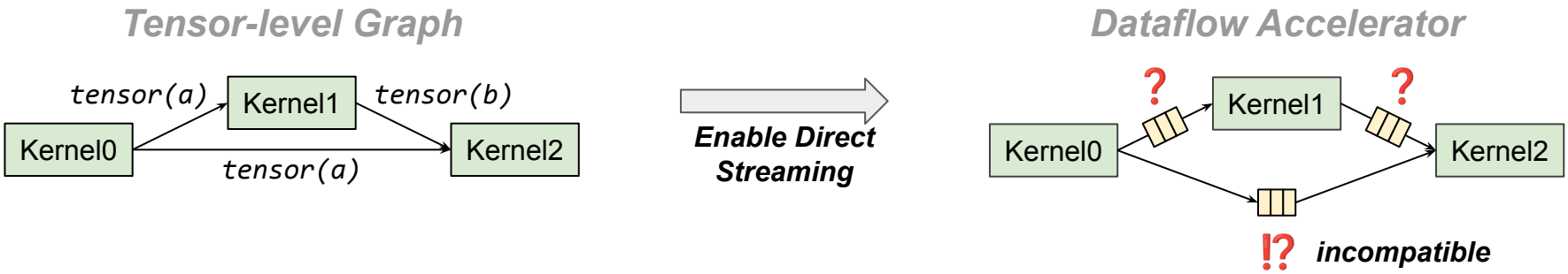


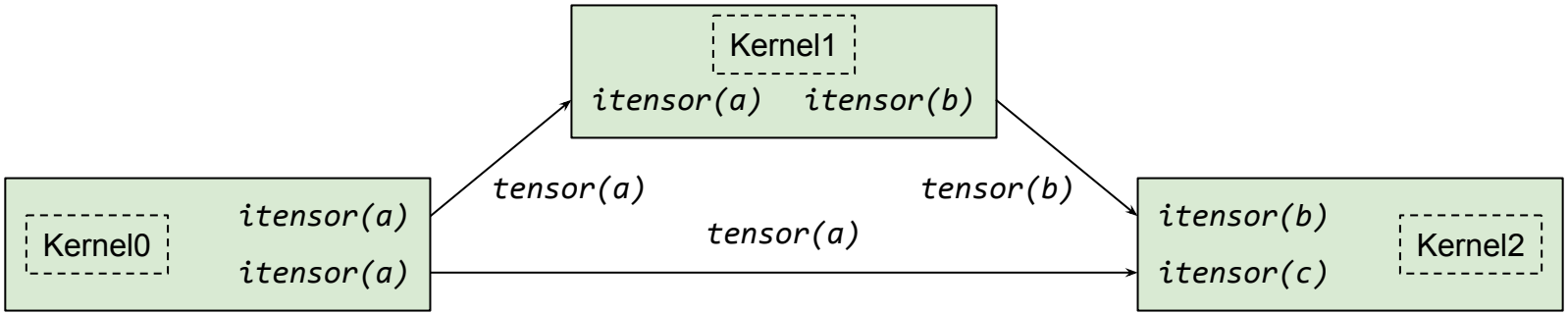
- Automate the generation of dataflow buffers, DMAs, and FIFOs
- Resolve pitfalls through systematic design space exploration
- Support auto-tuned or hand-written kernel integration

StreamTensor Outline

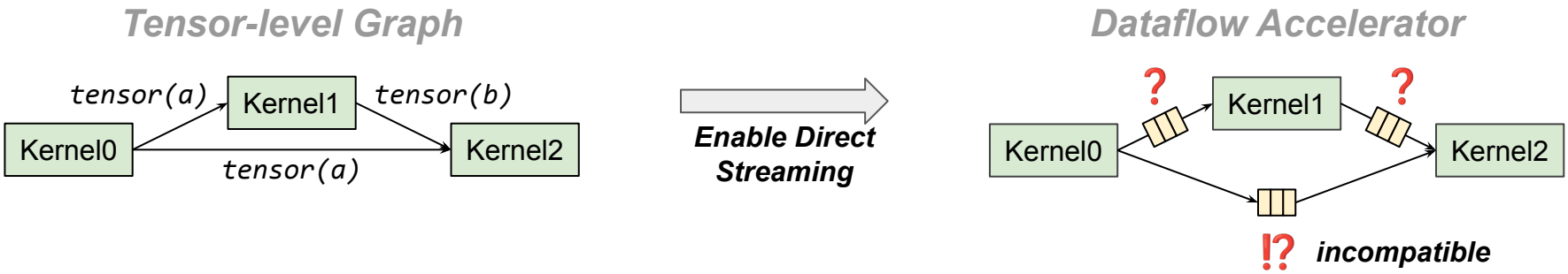
- Motivation
- StreamTensor Typing System
- StreamTensor Compilation Pipeline
- StreamTensor Design Spaces
- StreamTensor Results

Motivation of Iterative Tensor Type

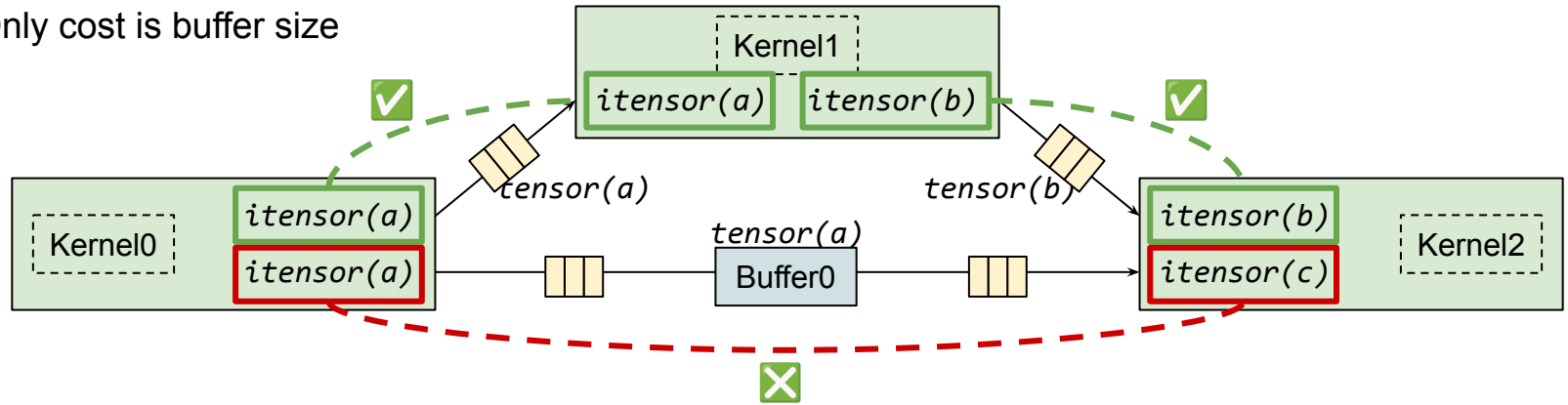




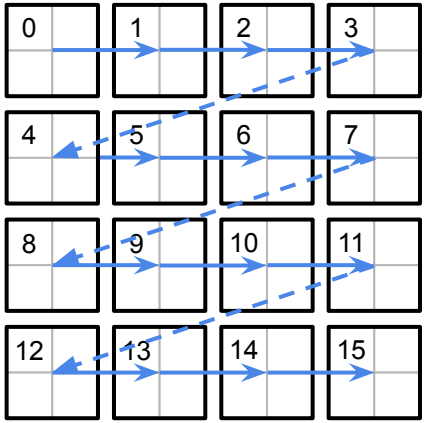
Motivation of Iterative Tensor Type (Cont.)



- By design, all kernels can be fused in a streaming manner
- Only cost is buffer size



Iterative Tensor Type



(a) itensor<2x2xf32,
iter_space: [4,4]*[2,2],
iter_map: (d0,d1)->(d0,d1)>

2x2xf32 = Element Shape & Type
iter_space = Iteration Space
= [Tripcounts]*[Steps]
iter_map = Iteration Affine Map

Iteration Space
[4,4]*[2,2]

[0,0]
[0,2]
[0,4]
[0,6]
[2,0]
[2,2]
…

Affine Mapping
(d0,d1)->(d0,d1)

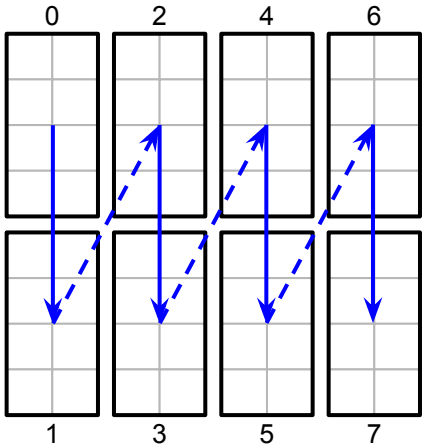
Element Shape
2x2

Data Space

[0,0]
[0,2]
[0,4]
[0,6]
[2,0]
[2,2]
…

Same `itensor` type means same stream layout/order

Iterative Tensor Type (Cont.)



(b) `itensor``<4x2xf32, iter_space: [4,2]*[2,4], iter_map: (d0,d1)->(d1,d0)>`

`4x2xf32` = Element Shape & Type

`iter_space` = Iteration Space
= [Tripcounts]*[Steps]

`iter_map` = Iteration Affine Map

Iteration Space
`[4,2]*[2,4]`

`[0,0]`
`[0,4]`
`[2,0]`
`[2,4]`
`[4,0]`
`[4,4]`
…

Affine Mapping
`(d0,d1)->(d1,d0)`

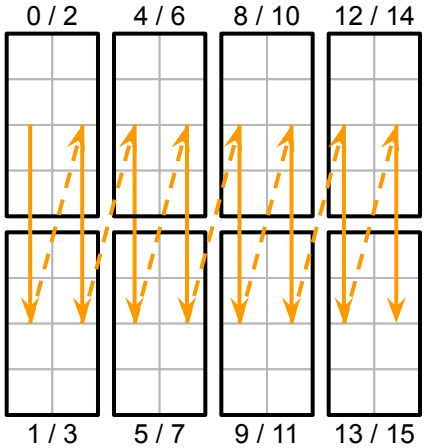
Element Shape
4x2

Data Space

`[0,0]`
`[4,0]`
`[0,2]`
`[4,2]`
`[0,4]`
`[4,4]`
…

Transposed stream layout/order

Iterative Tensor Type (Cont.)



(c) `itensor``<4x2xf32, iter_space: [4,2,2]*[2,1,4], iter_map: (d0,d1,d2)->(d2,d0)>`

`4x2xf32` = Element Shape & Type

`iter_space` = Iteration Space
= [Tripcounts]*[Steps]

`iter_map` = Iteration Affine Map

Iteration Space
`[4,2,2]*[2,1,4]`

`[0,0,0]`
`[0,0,4]`
`[0,1,0]`
`[0,1,4]`
`[2,0,0]`
`[2,0,4]`
…

Affine Mapping
`(d0,d1,d2)->(d2,d0)`

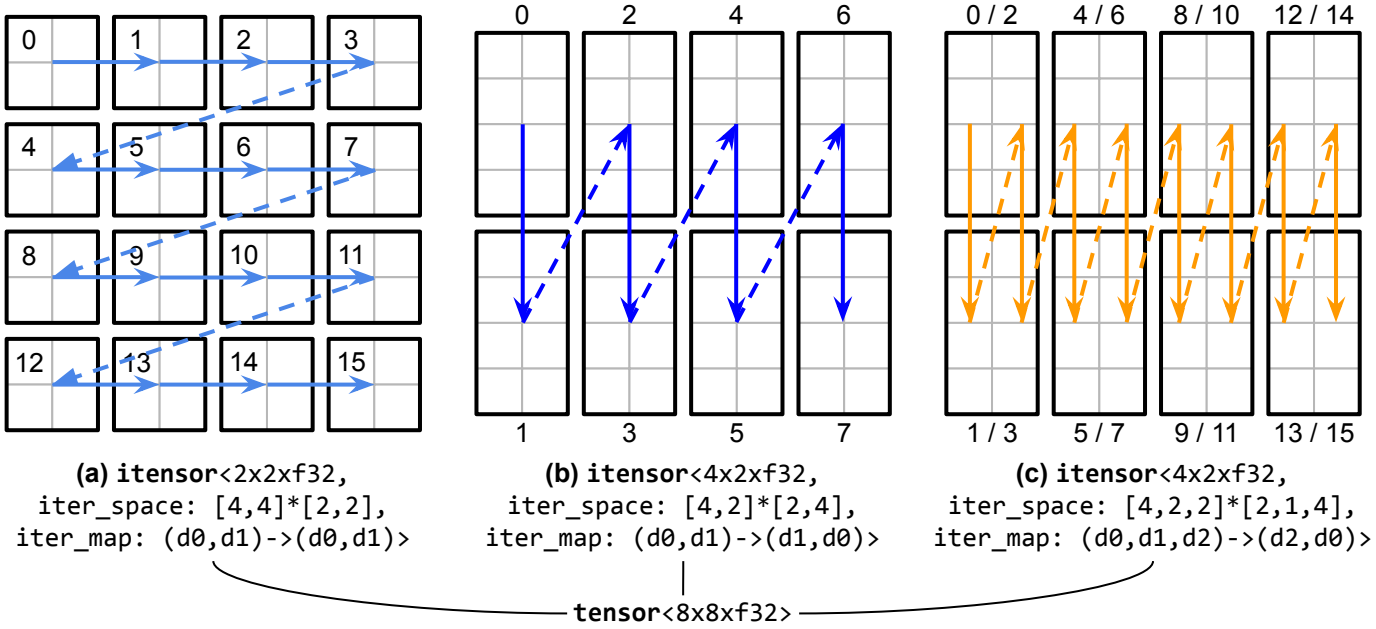
Element Shape
4x2

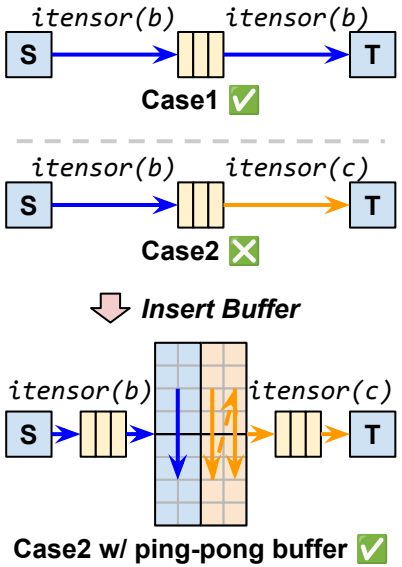
Data Space

`[0,0]`
`[4,0]`
`[0,0]`
`[4,0]`
`[0,2]`
`[4,2]`
…

Transposed and repeated stream layout/order

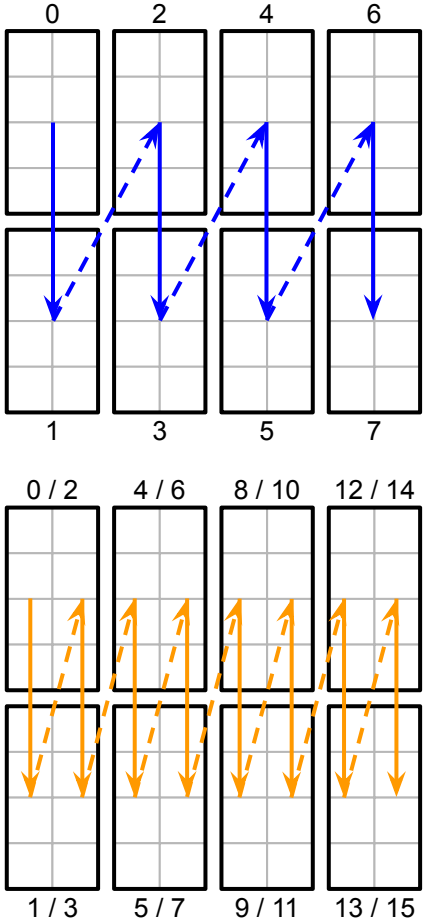
Iterative Tensor Type (Cont.)





The minimal buffer size is inferred from `itensor` types, which are used as "cost" during kernel fusion

Infer Buffer Shape from Iterative Tensor Types



(b) `itensor<4x2xf32,`

`iter_space: [4,2]*[2,4],`

`iter_map: (d0,d1)->(d1,d0)>`

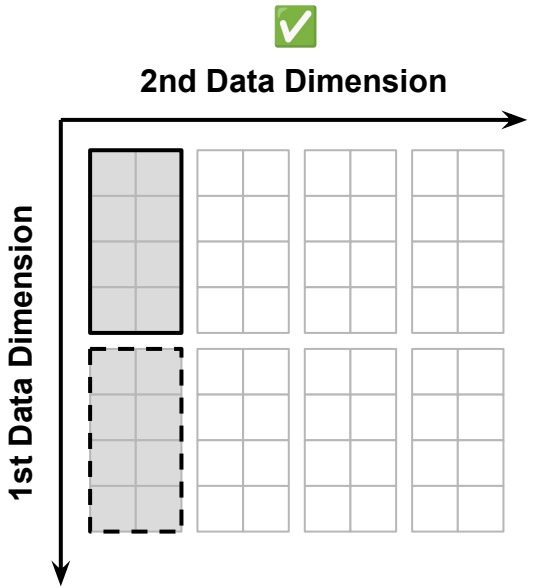
Element Shape ✅

Iteration Dimension ✅

(c) `itensor<4x2xf32,`

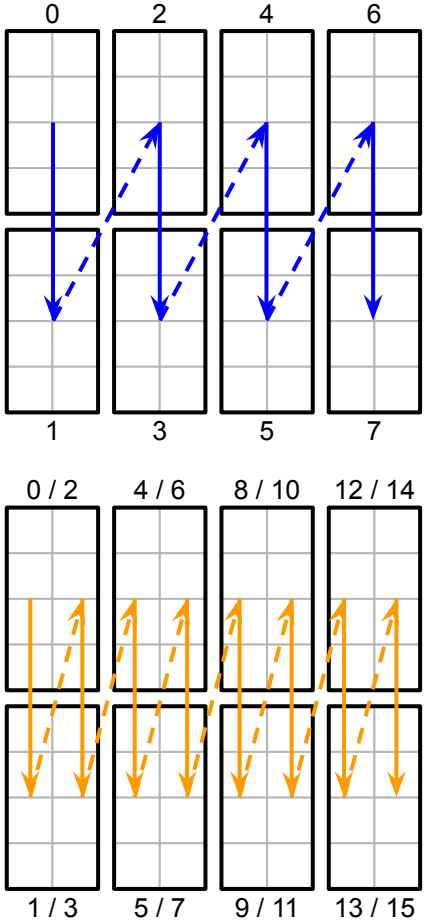
`iter_space: [4,2,2]*[2,1,4],`

`iter_map: (d0,d1,d2)->(d2,d0)>`



Traverse each data dimension from little-endian

Infer Buffer Shape from Iterative Tensor Types (Cont.)



(b) `itensor<4x2xf32,`

`iter_space: [4,2]*[2,4],`

`iter_map: (d0,d1)->(d1,d0)>`

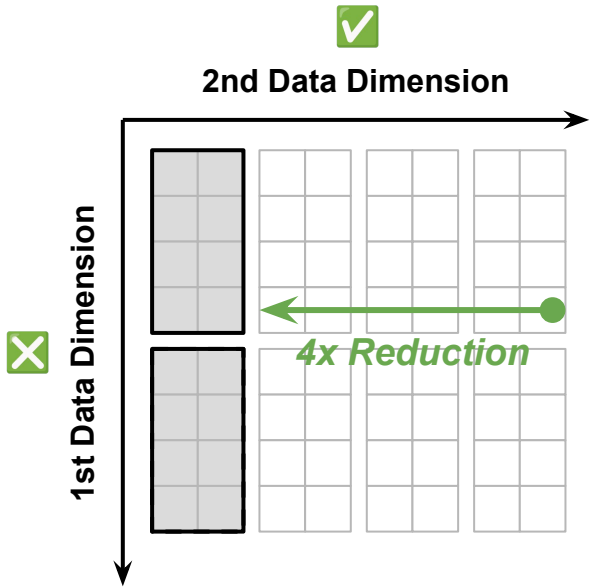
Element Shape ✅

Iteration Dimension ❌

(c) `itensor<4x2xf32,`

`iter_space: [4,2,2]*[2,1,4],`

`iter_map: (d0,d1,d2)->(d2,d0)>`

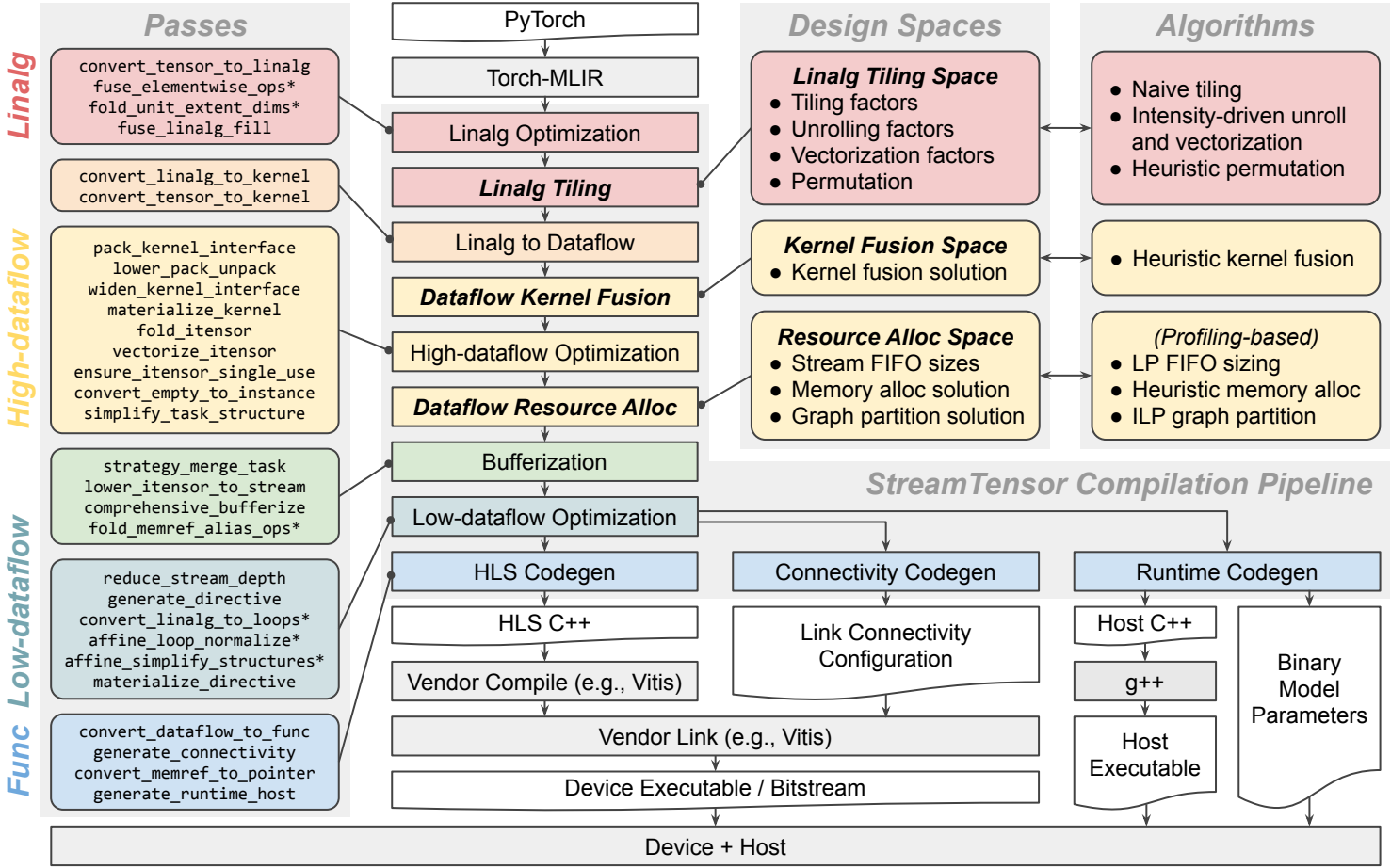


Traverse each data dimension from little-endian

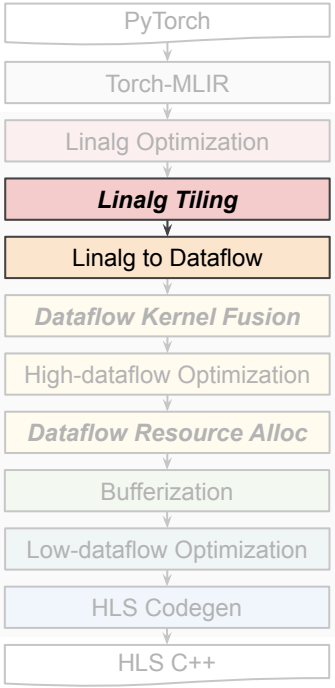
StreamTensor Outline

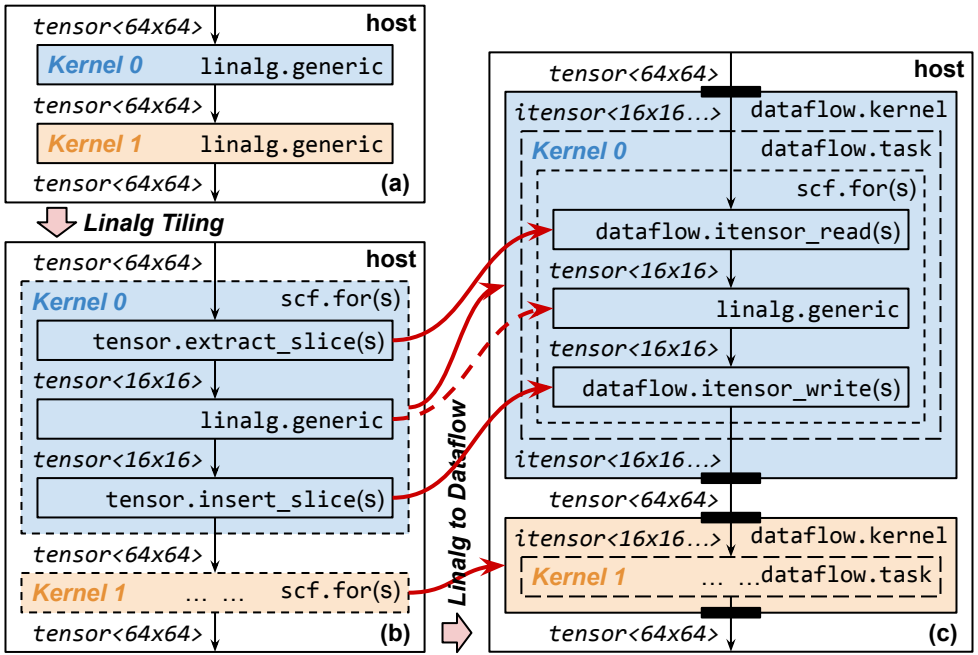
- Motivation
- StreamTensor Typing System
- StreamTensor Compilation Pipeline
- StreamTensor Design Spaces
- StreamTensor Results

StreamTensor Framework Overview



Linalg Tiling + Linalg-to-Dataflow Conversion





`scf` = Structured Control Flow

`linalg` = Linear Algebra

Transformed

Unchanged

`dataflow.kernel`

- Isolated from above
- Convert tensor to/from itensor at the boundary, representing DMAs implicitly
- Contains a graph of tasks
- *Easy for kernel fusion*

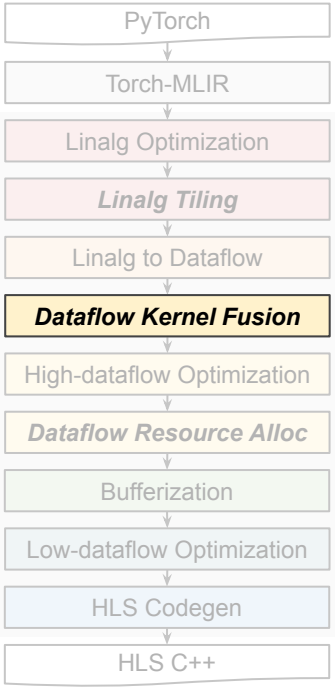
`dataflow.task`

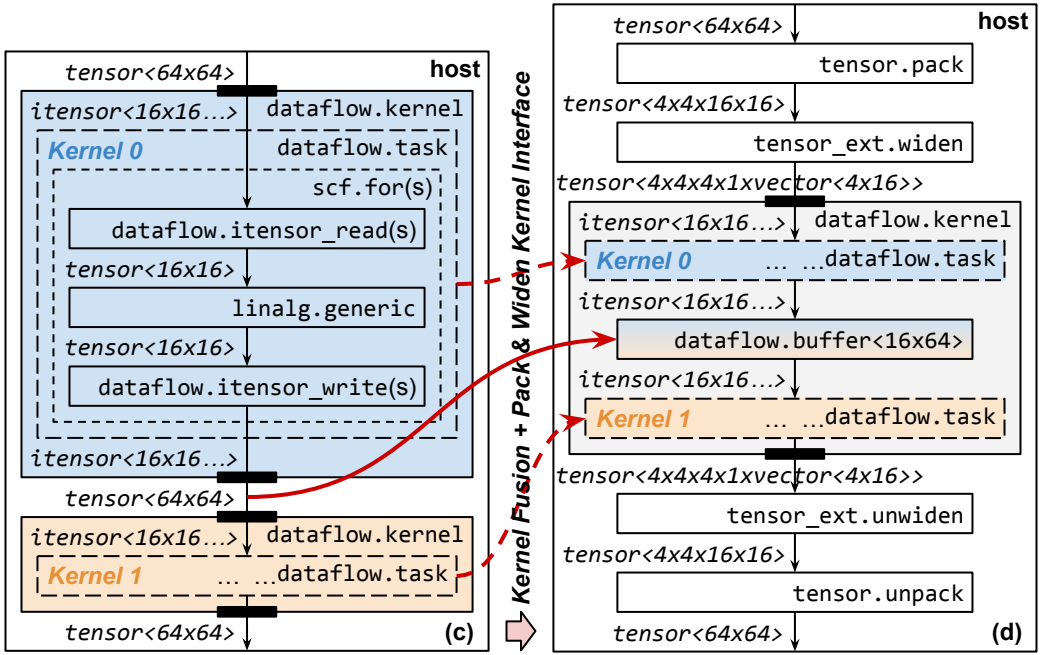
- Transparent from above
- Can be nested to represent a hierarchy of dataflow
- Contains a graph of operations
- Easy for task manipulation

`dataflow.itensor_read/write`

- Read a tensor slice/tile from an itensor (FIFO pull)
- Write a tensor slice/tile into an itensor (FIFO push)

Kernel Fusion + Pack & Widen Kernel Interface





`scf` = Structured Control Flow

`linalg` = Linear Algebra

Transformed

Unchanged

`dataflow.buffer`

- Convert an input itensor to another itensor with different layout

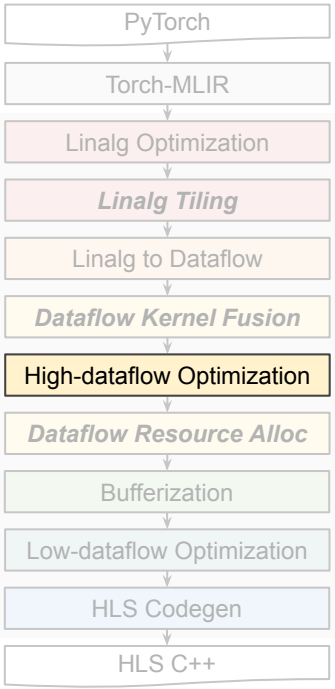
`tensor.pack/unpack`

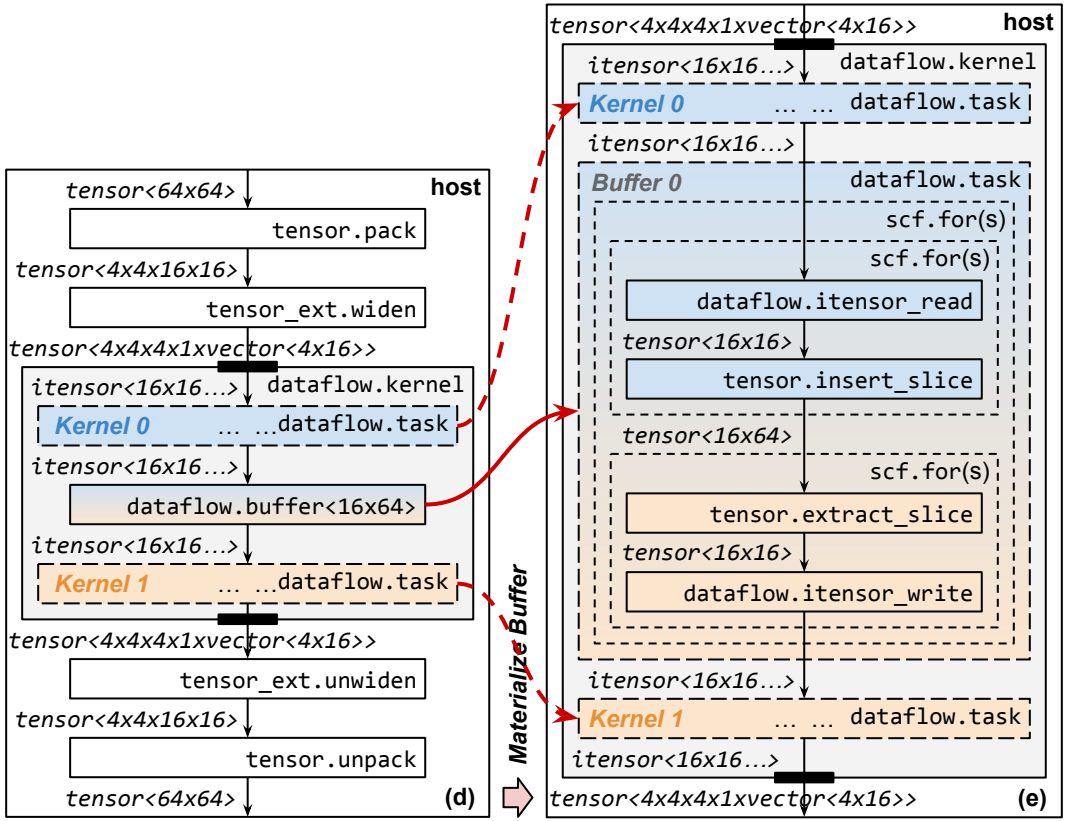
- Pack tensor from normal layout to tiled layout
- *Improve external memory access efficiency*

`tensor_ext.widen/unwiden`

- Widen tensor from scalar element to vector element
- *Improve external memory access bitwidth*

Buffer Materialization





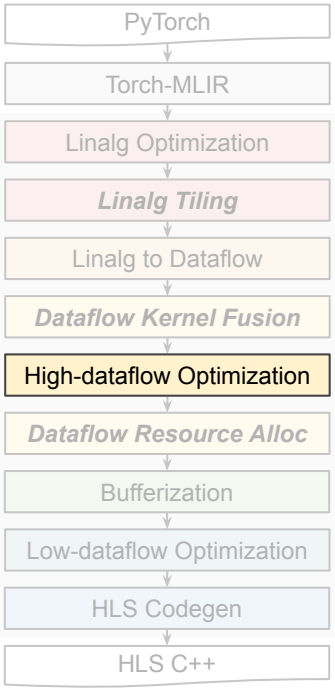
Before Materialization

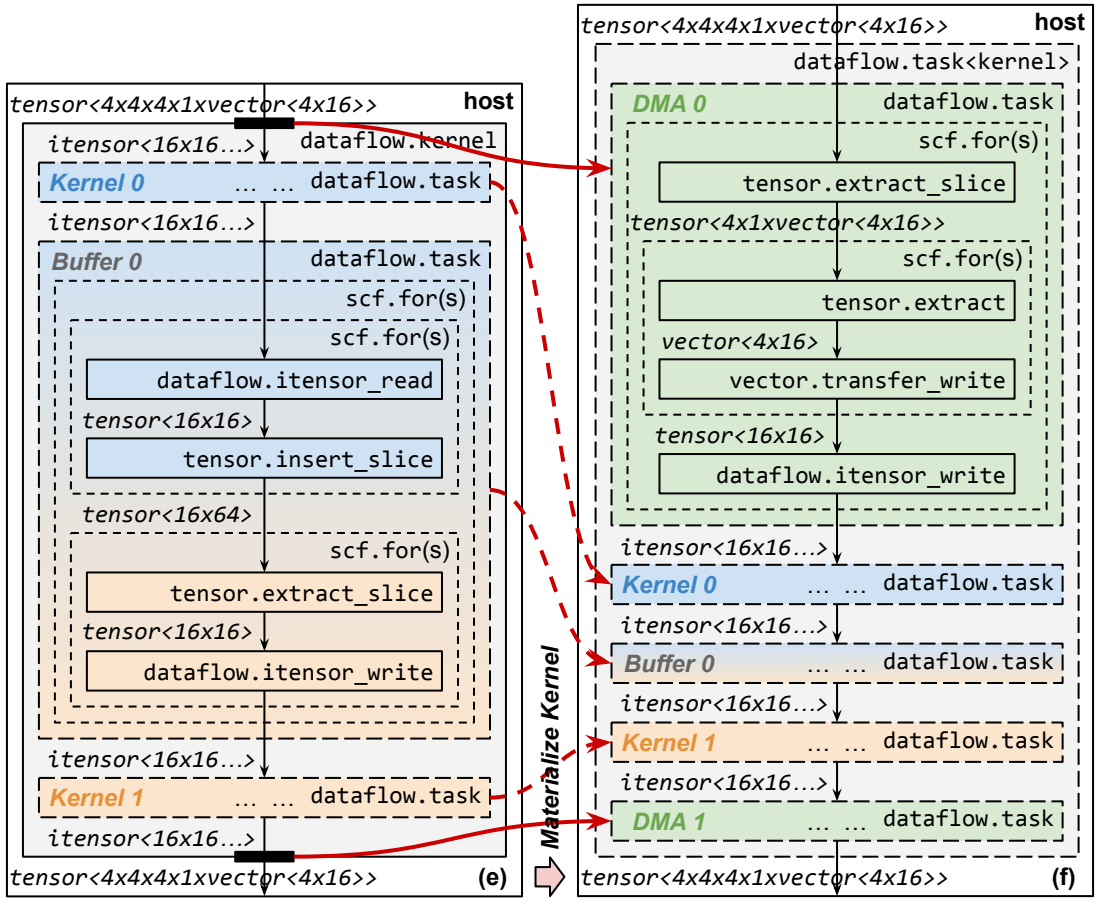
- Buffer is represented by a single operation
- *Easy for optimizations like CSE*

After Materialization

- Buffer is represented by task, loops, and tensor or itensor operations
- *Easy for subsequent dataflow optimizations*

Kernel Materialization





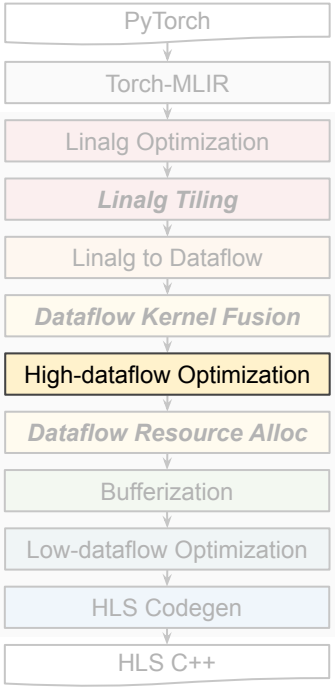
Before Materialization

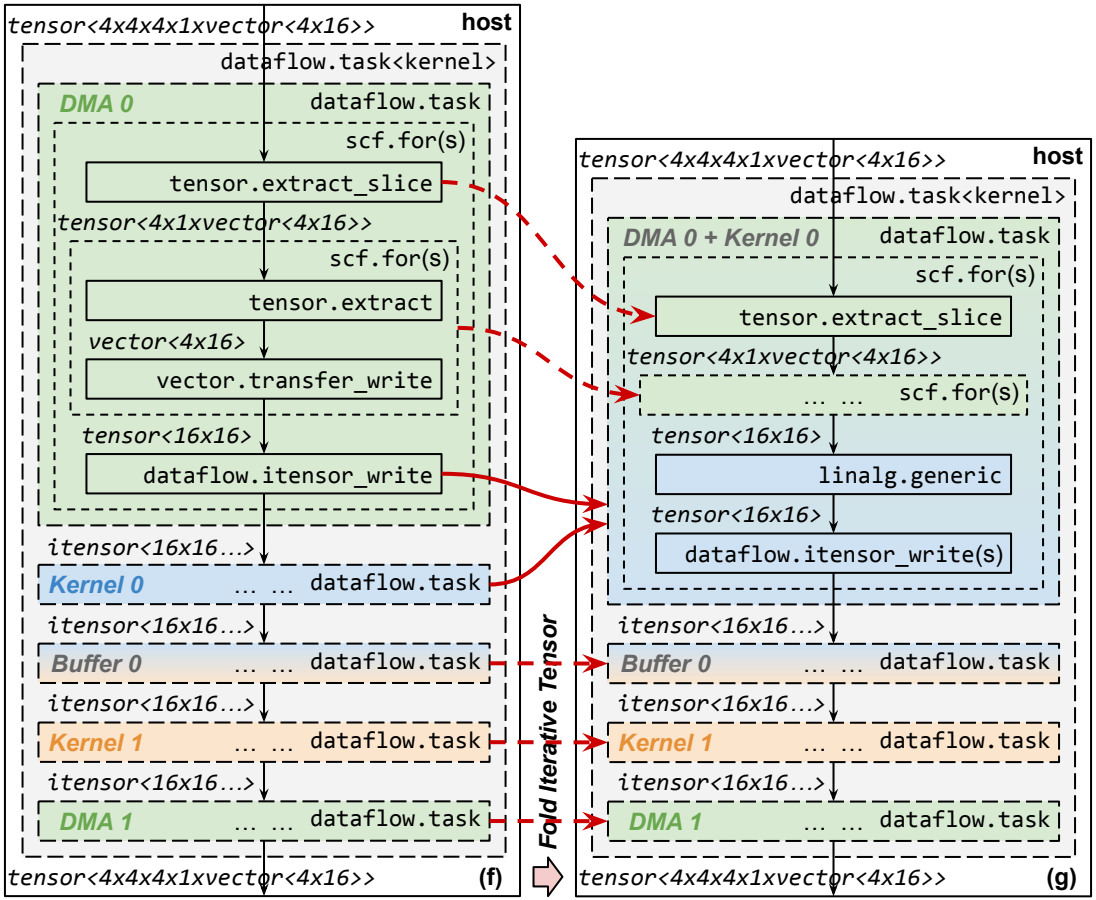
- DMA is represented implicitly at boundary of kernels
- *Easy for kernel fusion*

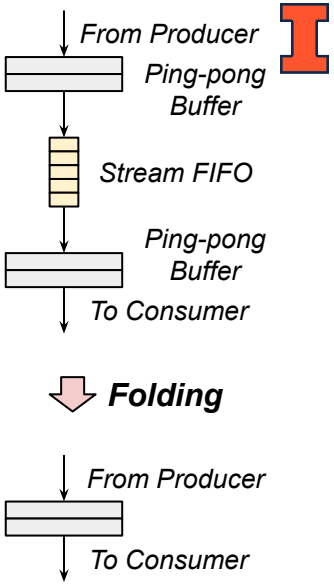
After Materialization

- DMA is represented by task, loops, and tensor or itensor operations
- *Easy for subsequent dataflow optimizations*

Fold Iterative Tensor



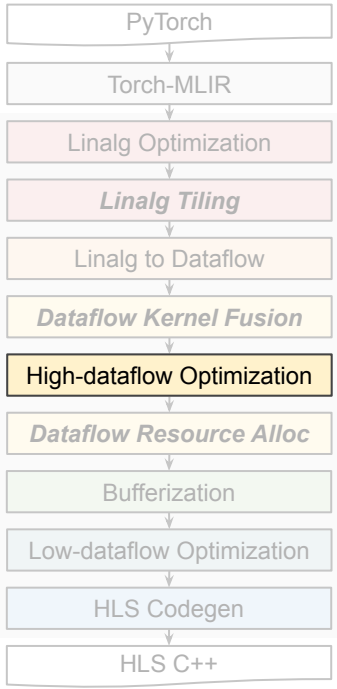


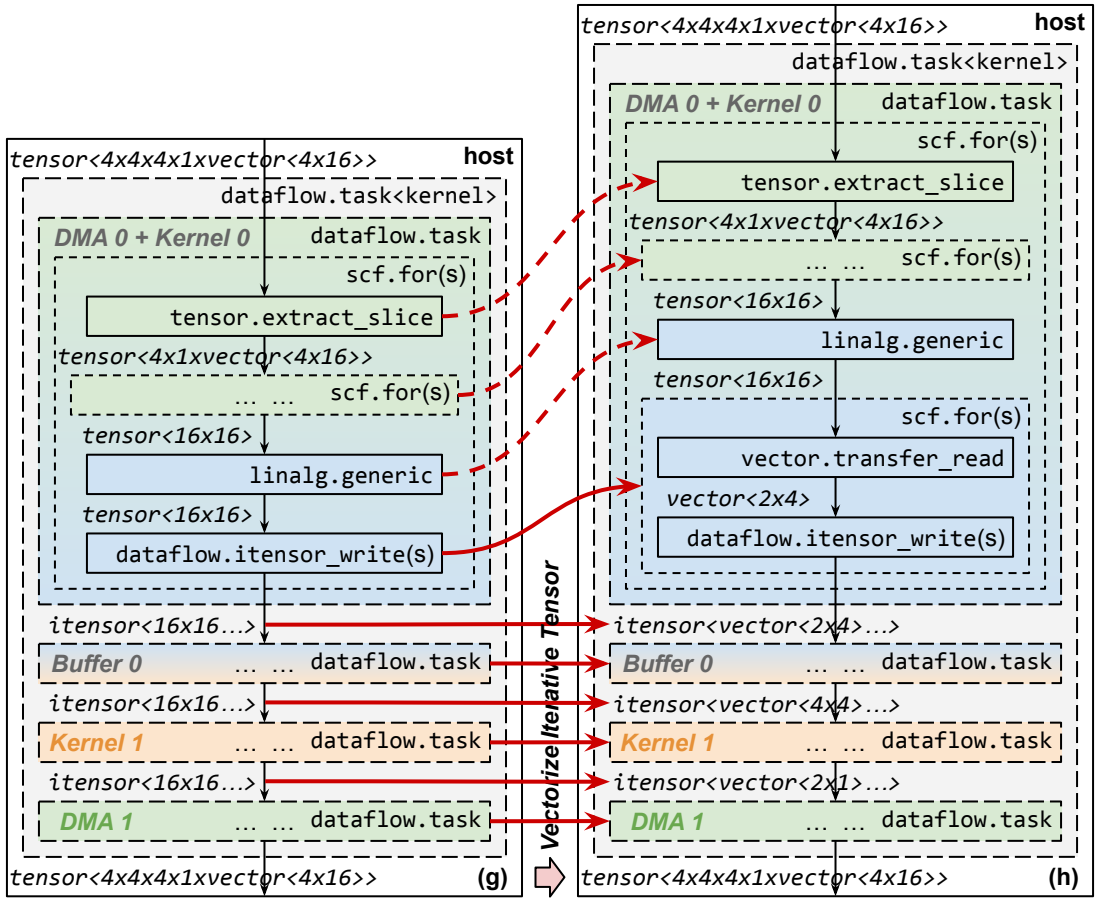


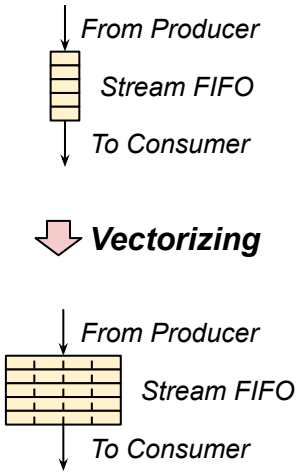
Benefits

- Merge two local ping-pong buffers into one, reducing on-chip memory
- Increase the overlap between the tasks, reducing latency

Vectorize Iterative Tensor







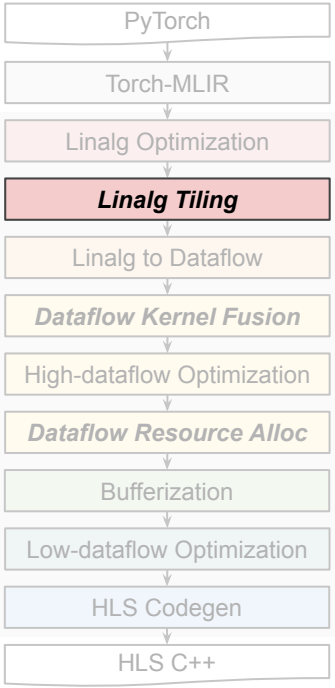
Benefits

- Increase the bitwidth of FIFOs
- Match computation parallelization
- Match external memory bandwidth

StreamTensor Outline

- Motivation
- StreamTensor Typing System
- StreamTensor Compilation Pipeline
- StreamTensor Design Spaces
- StreamTensor Results

Linalg Tiling Space




```
1  float A[32][16];
2  NODE0_I: for (int i=0; i<32; i++)
3    NODE0_K: for (int k=0; k<16; k++)
4      A[i][k] = ...; // Load array A.
5
6  float B[16][16];
7  NODE1_K: for (int k=0; k<16; k++)
8    NODE1_J: for (int j=0; j<16; j++)
9      B[k][j] = ...; // Load array B.
10
11 float C[16][16];
12 NODE2_I: for (int i=0; i<16; i++)
13   NODE2_J: for (int j=0; j<16; j++)
14     NODE2_K: for (int k=0; k<16; k++)
15       C[i][j] = A[i*2][k] * B[k][j];
```

HIDA IA+CA Algorithm

Node	Intensity	Parallel Factor		Loop Unroll Factors			
		w/o IA	w/ IA	IA+CA	IA	CA	Naive
Node0	512	32	4	[4, 1]	[2, 2]	[8, 4]	[4, 8]
Node1	256	32	2	[1, 2]	[1, 2]	[4, 8]	[4, 8]
Node2	4,096	32	32	[4, 8, 1]	[4, 8, 1]	[4, 8, 1]	[4, 8, 1]

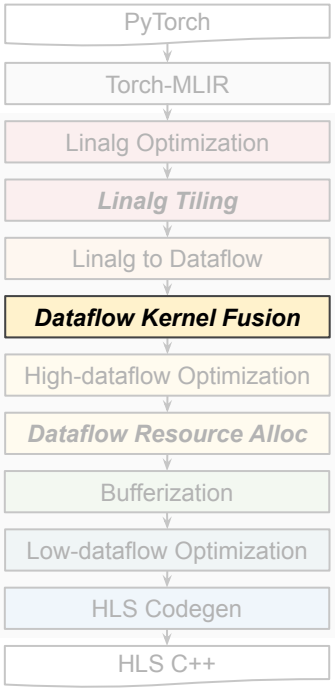
Intensity-aware (IA)
Connectedness-aware (CA)
HIDA DSE

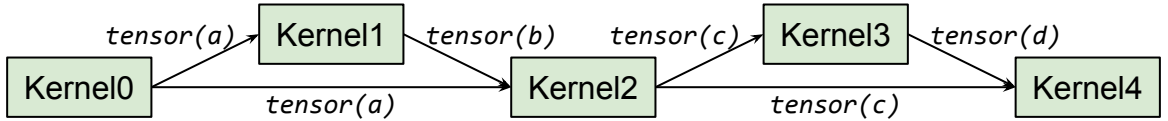
Naive
ScaleHLS
DSE

Array	Array Partition Factors				Bank Number				
	IA+CA	IA	CA	Naive	IA+CA	IA	CA	Naive	
A	[8, 1]	[8, 2]	[8, 4]	[8, 8]	8	16	32	64	*8x*
B	[1, 8]	[2, 8]	[4, 8]	[8, 8]	8	16	32	64	*8x*
C	[4, 8]	[4, 8]	[4, 8]	[4, 8]	32	32	32	32	*1x*

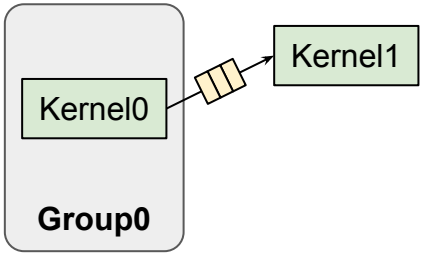
The design space is exposed at Python level, open for auto-tuning or other optimization algorithm

Kernel Fusion Space





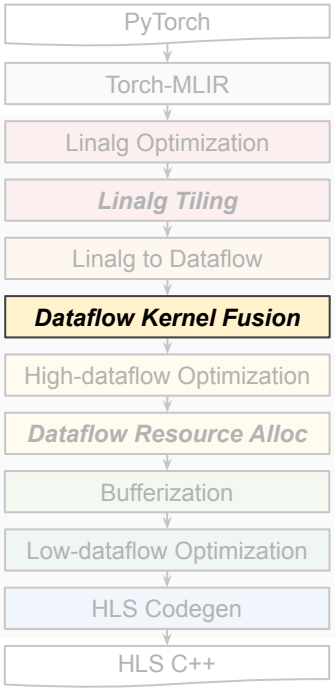
Topological Order Traversal

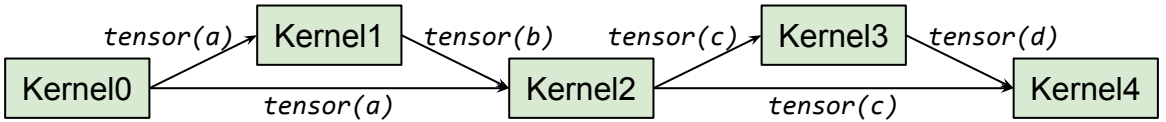


Group0 On-chip Memory Available

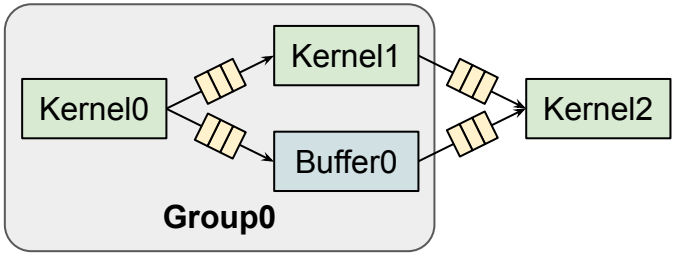
	Group0	**Group1**
Kernels	Kernel0	
Cost	$\text{Local}_{\text{Kernel0}}$	

Kernel Fusion Space (Cont.)





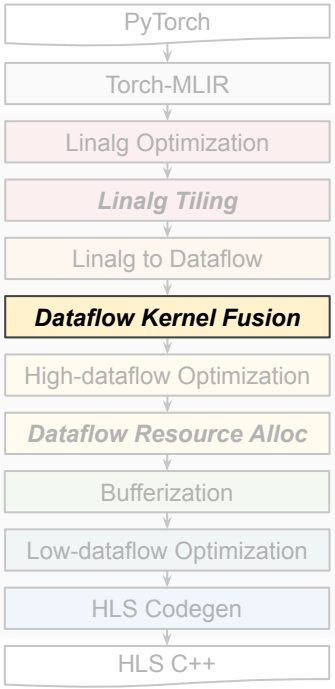
Topological Order Traversal

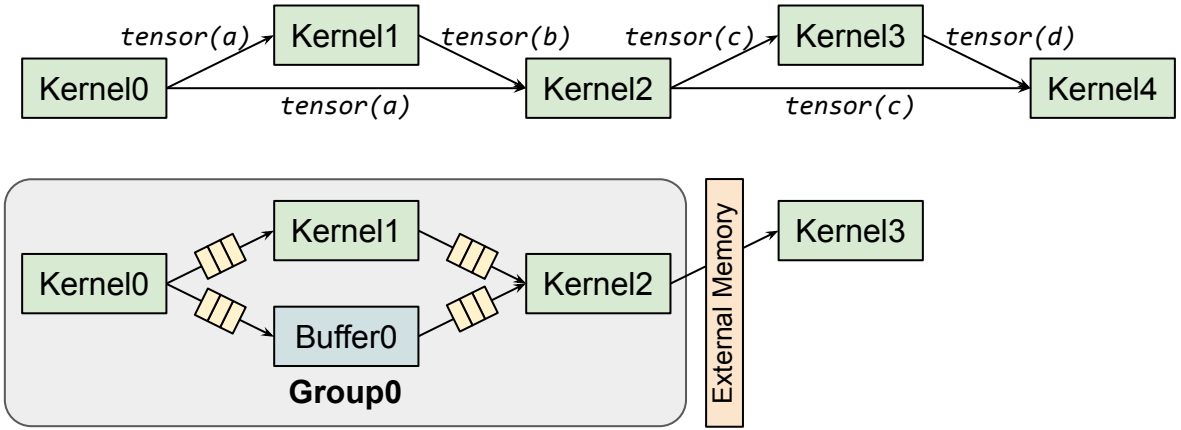


Group0 On-chip Memory Available

	Group0	Group1
Kernels	Kernel0 Kernel1	
Cost	$\text{Local}_{\text{Kernel0}}$ $\text{Local}_{\text{Kernel1}}$	

Kernel Fusion Space (Cont.)



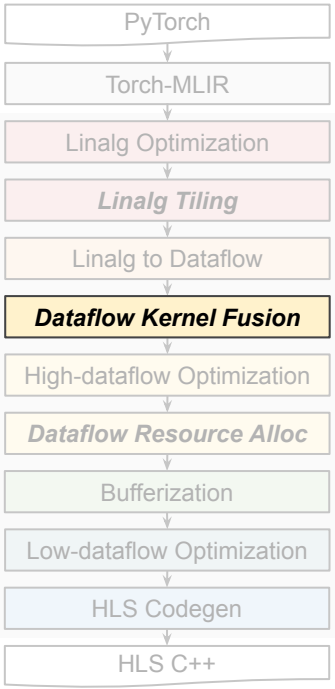


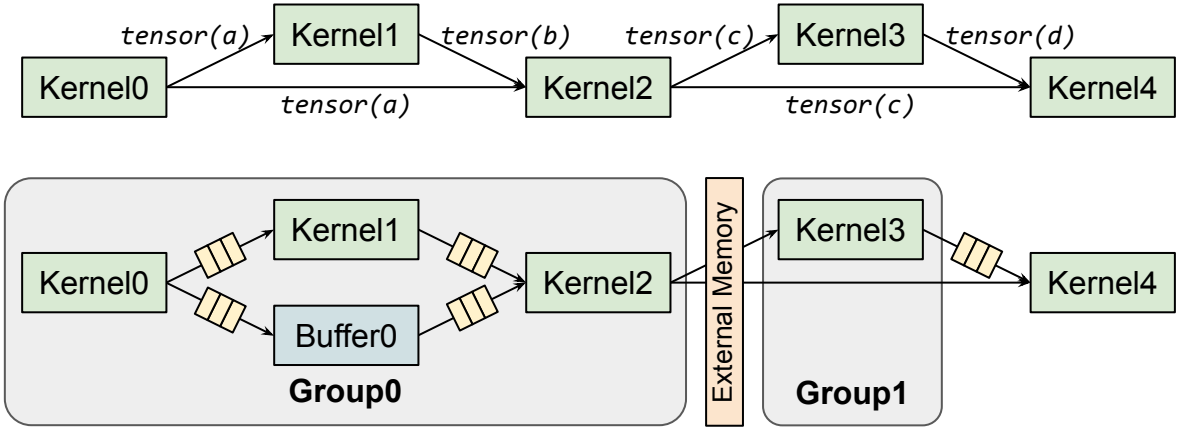
Topological Order Traversal

Group0 On-chip Memory Full

	Group0	Group1
Kernels	Kernel0 Kernel1 Kernel2	
Cost	$Local_{Kernel0}$ $Local_{Kernel1}$ Buffer0 $Local_{Kernel2}$	

Kernel Fusion Space (Cont.)



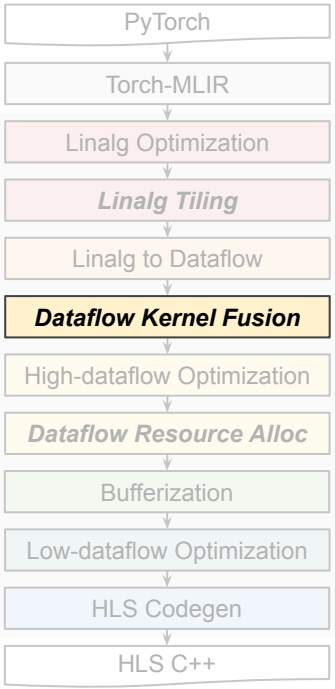


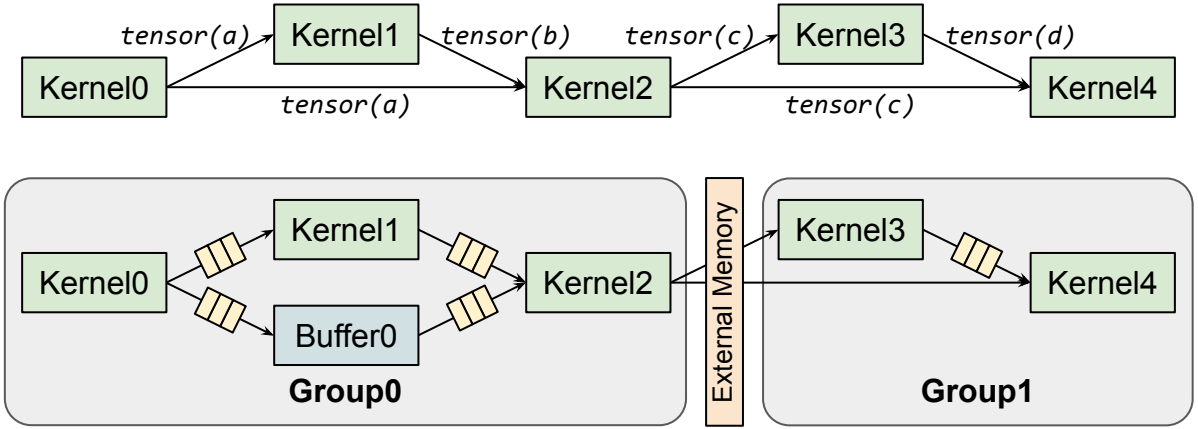
Topological Order Traversal

Select the Nearest Predecessor

	Group0	Group1
Kernels	Kernel0 Kernel1 Kernel2	Kernel3
Cost	$Local_{Kernel0}$ $Local_{Kernel1}$ Buffer0 $Local_{Kernel2}$	$Local_{Kernel3}$

Kernel Fusion Space (Cont.)





Topological Order Traversal

Intra-group: Streaming

Inter-group: External Memory

	Group0	Group1
Kernels	Kernel0 Kernel1 Kernel2	Kernel3 Kernel4
Cost	$Local_{Kernel0}$ $Local_{Kernel1}$ Buffer0 $Local_{Kernel2}$	$Local_{Kernel3}$ $Local_{Kernel4}$

Each Group

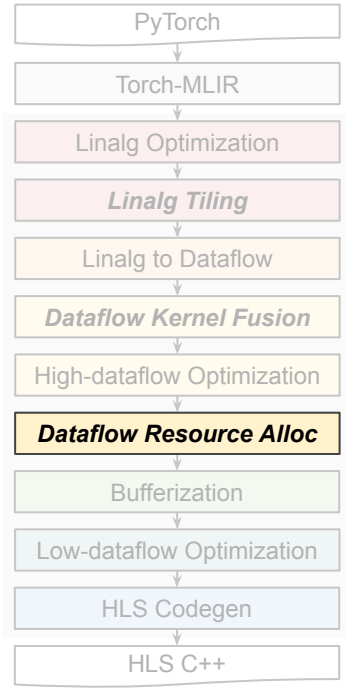
⬇️

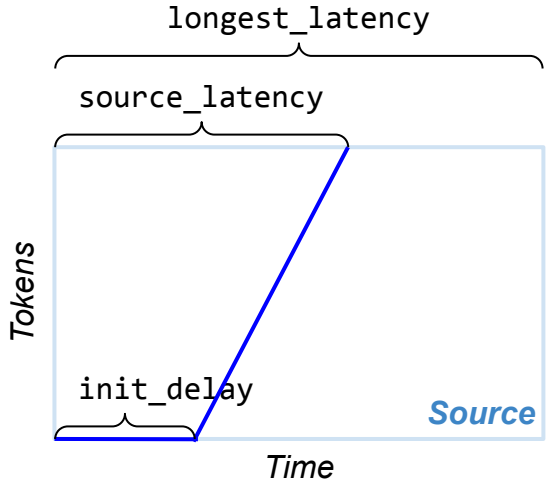
One Kernel

⬇️

One Bitstream (if on FPGA)

Resource Allocation Space - FIFO Sizing

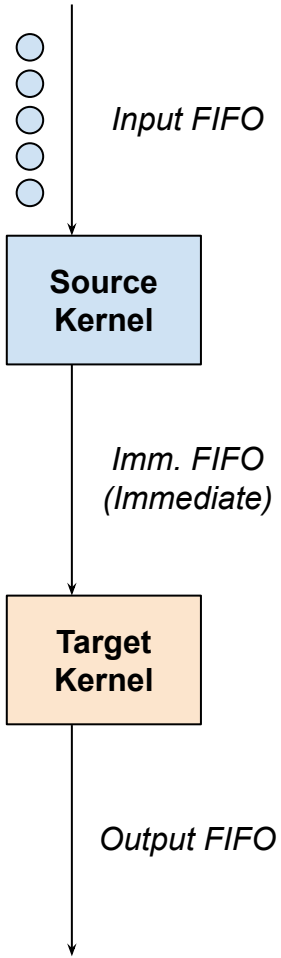




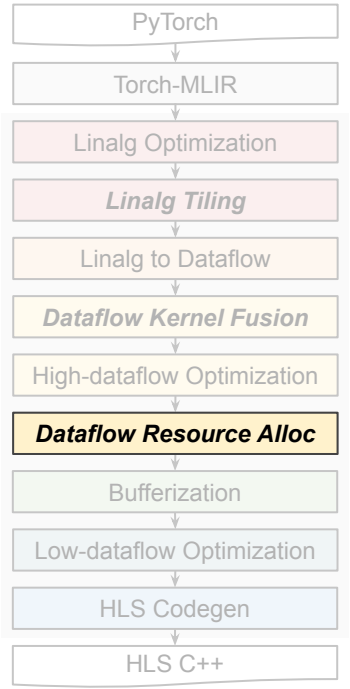
Source Token Production Curve (Push Imm. FIFO)

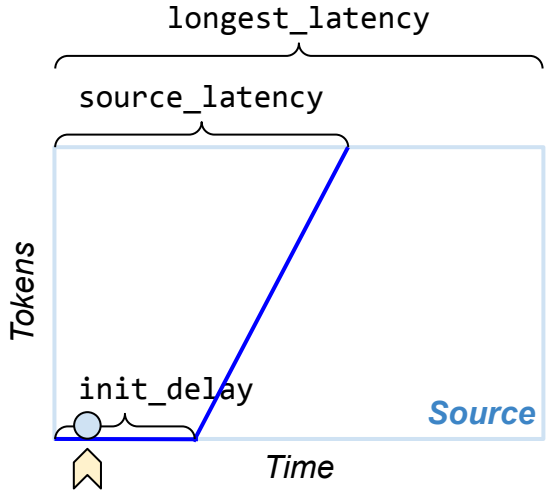
`token` = Atomic element passed between dataflow kernels

`longest_latency` = The longest latency among all the dataflow kernels in the accelerator, determining the maximum throughput



Token Number Estimation Heuristic

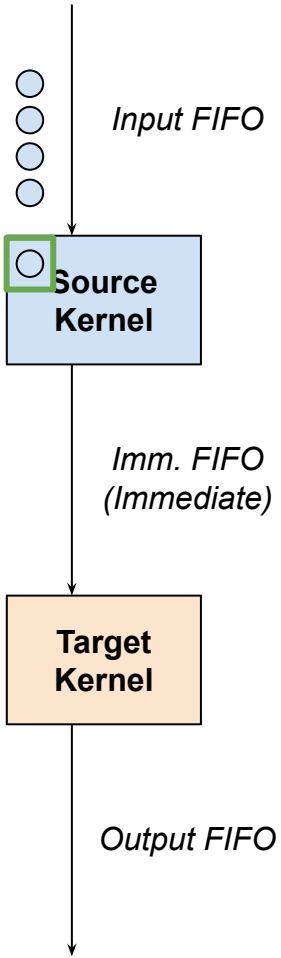




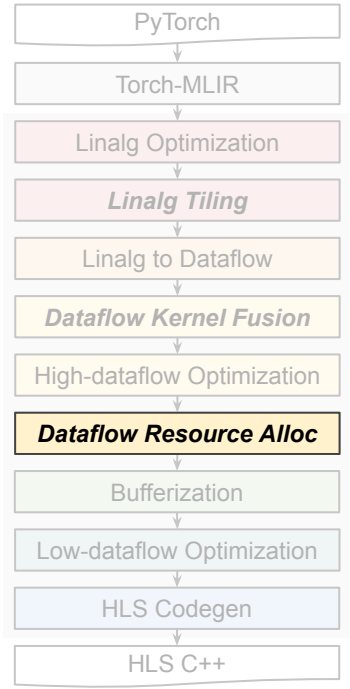
Source Token Production Curve (Push Imm. FIFO)

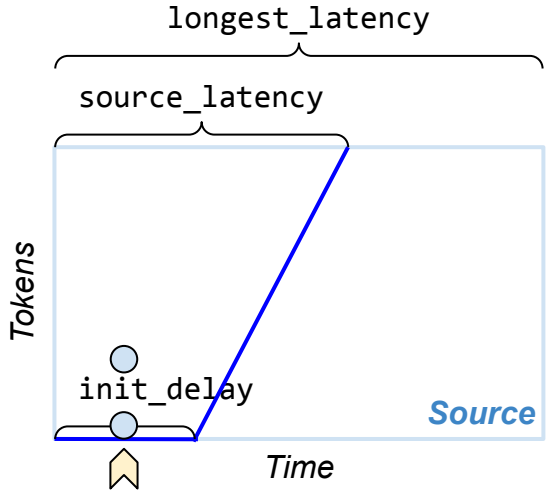
`token` = Atomic element passed between dataflow kernels

`longest_latency` = The longest latency among all the dataflow kernels in the accelerator, determining the maximum throughput



Token Number Estimation Heuristic (Cont.)

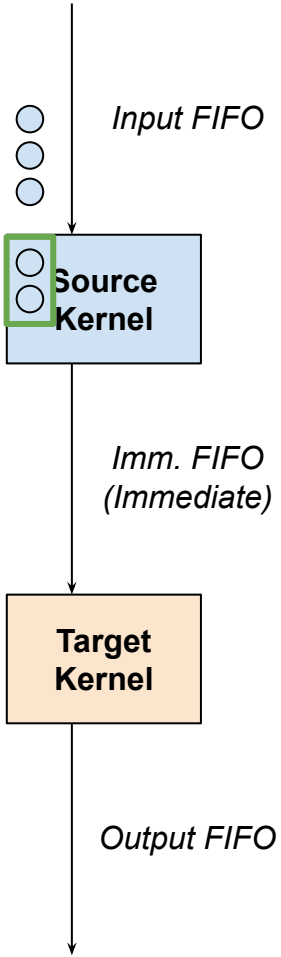




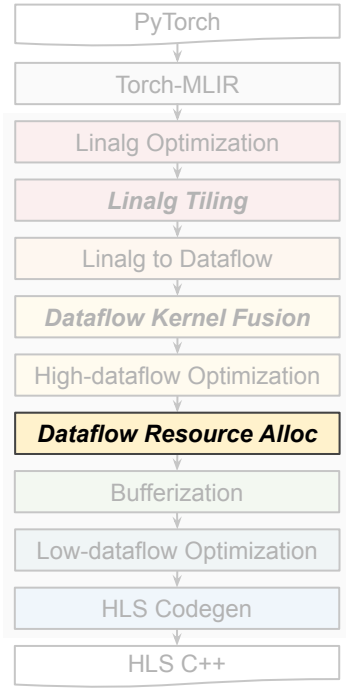
Source Token Production Curve (Push Imm. FIFO)

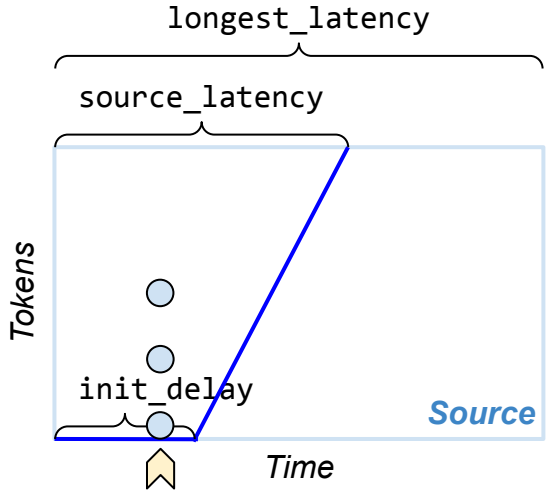
`token` = Atomic element passed between dataflow kernels

`longest_latency` = The longest latency among all the dataflow kernels in the accelerator, determining the maximum throughput



Token Number Estimation Heuristic (Cont.)

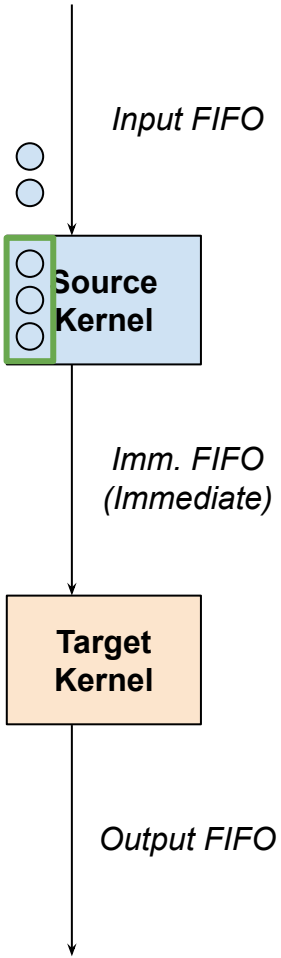




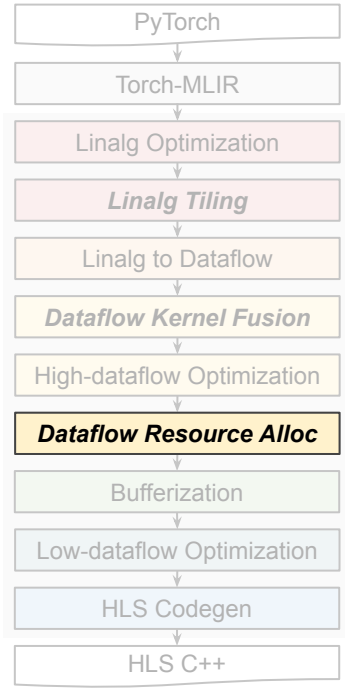
Source Token Production Curve (Push Imm. FIFO)

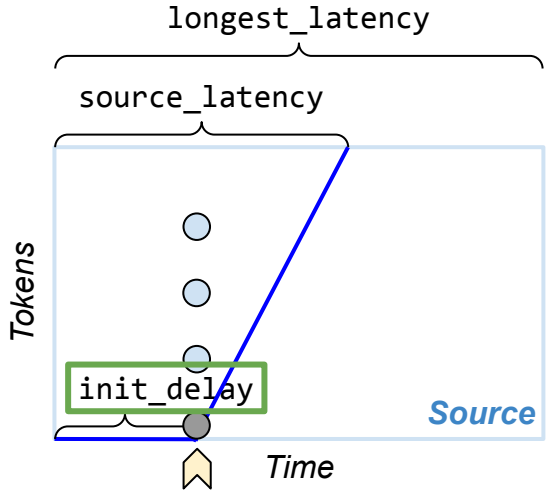
`token` = Atomic element passed between dataflow kernels

`longest_latency` = The longest latency among all the dataflow kernels in the accelerator, determining the maximum throughput



Token Number Estimation Heuristic (Cont.)

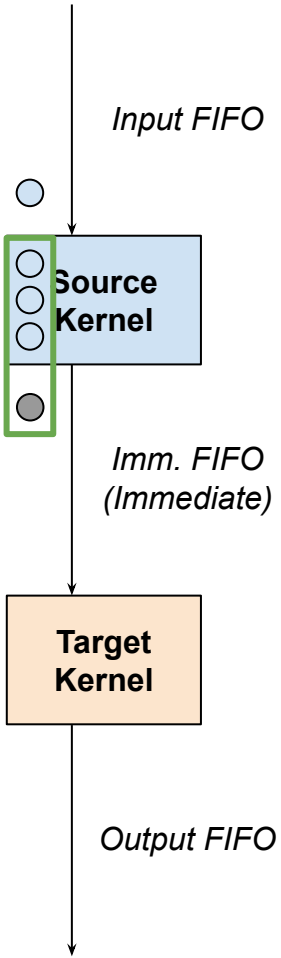




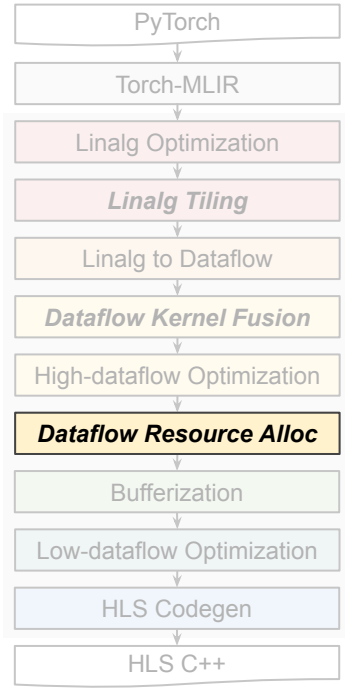
Source Token Production Curve (Push Imm. FIFO)

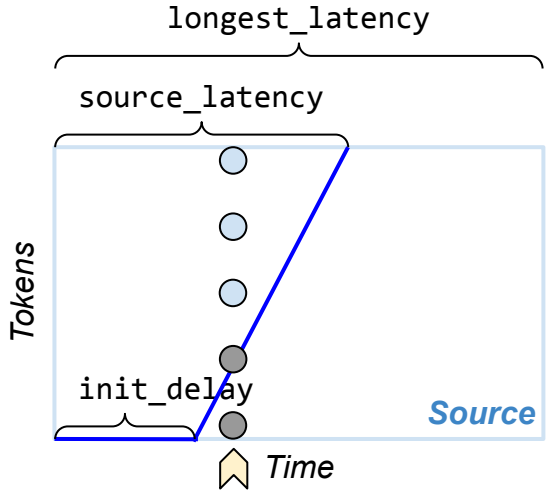
`token` = Atomic element passed between dataflow kernels

`longest_latency` = The longest latency among all the dataflow kernels in the accelerator, determining the maximum throughput



Token Number Estimation Heuristic (Cont.)

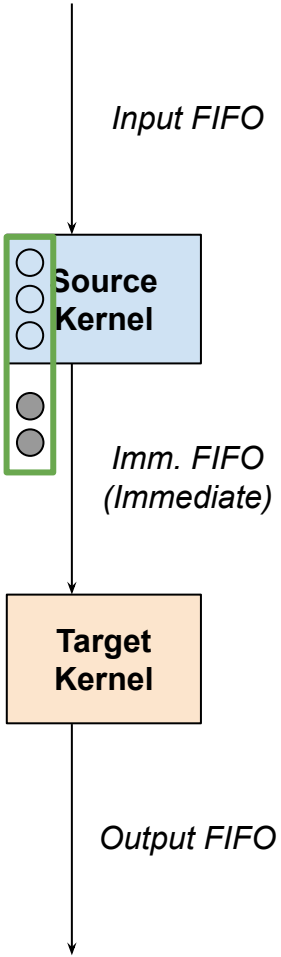




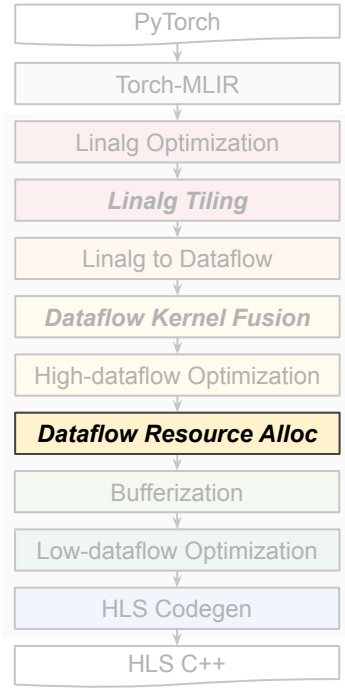
Source Token Production Curve (Push Imm. FIFO)

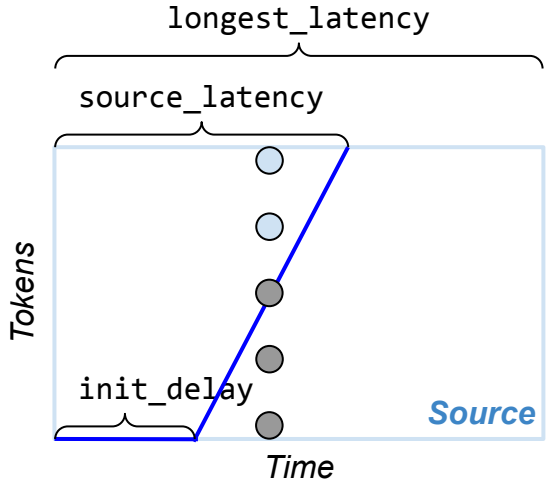
`token` = Atomic element passed between dataflow kernels

`longest_latency` = The longest latency among all the dataflow kernels in the accelerator, determining the maximum throughput



Token Number Estimation Heuristic (Cont.)

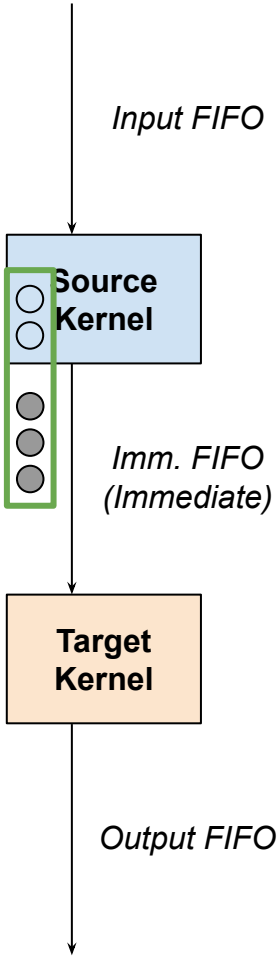




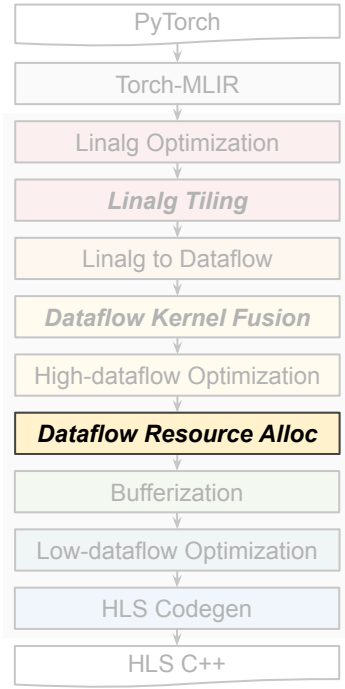
Source Token Production Curve (Push Imm. FIFO)

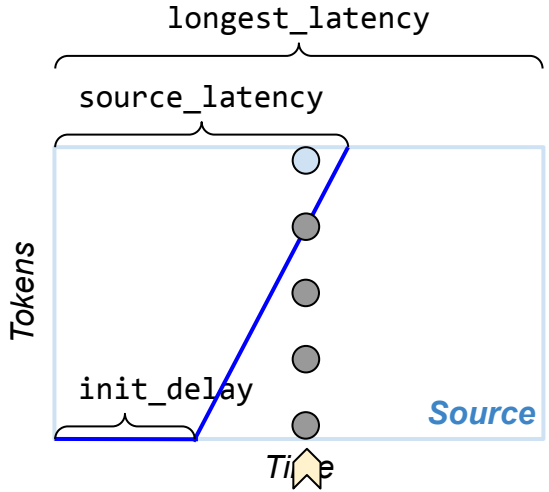
`token` = Atomic element passed between dataflow kernels

`longest_latency` = The longest latency among all the dataflow kernels in the accelerator, determining the maximum throughput



Token Number Estimation Heuristic (Cont.)

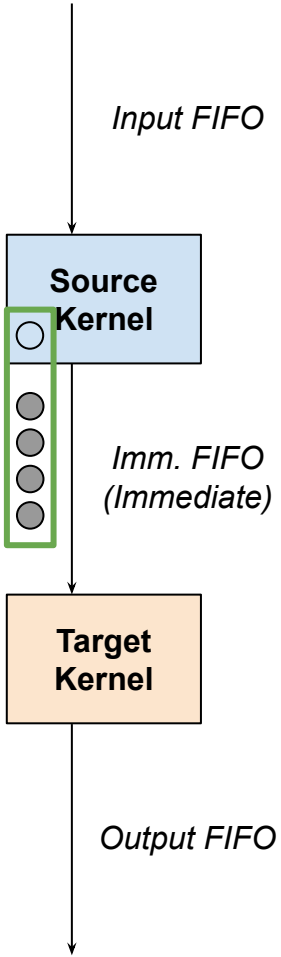




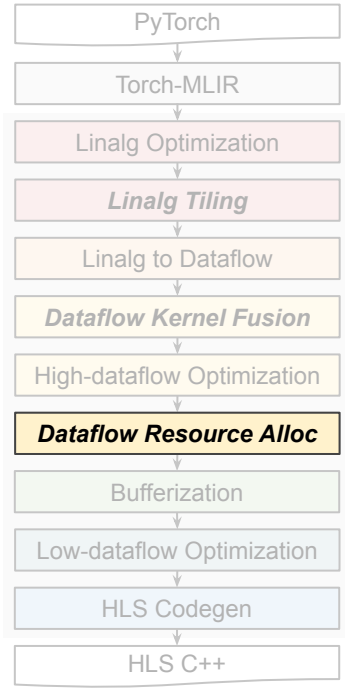
Source Token Production Curve (Push Imm. FIFO)

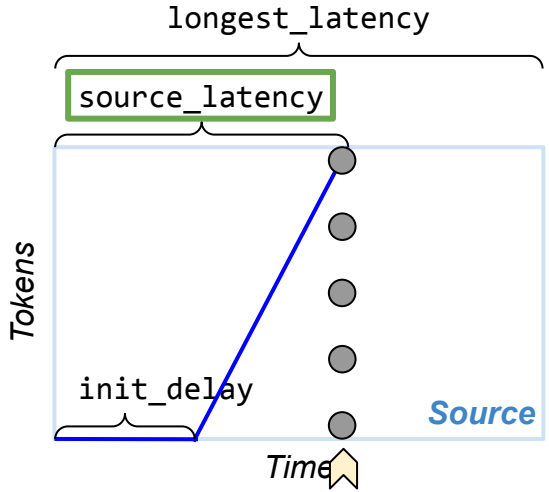
`token` = Atomic element passed between dataflow kernels

`longest_latency` = The longest latency among all the dataflow kernels in the accelerator, determining the maximum throughput



Token Number Estimation Heuristic (Cont.)

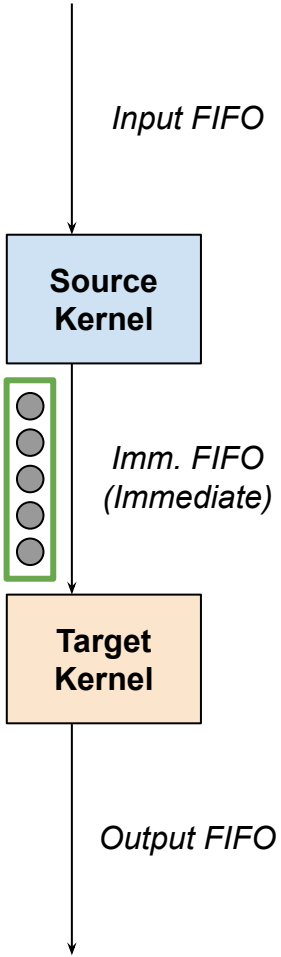




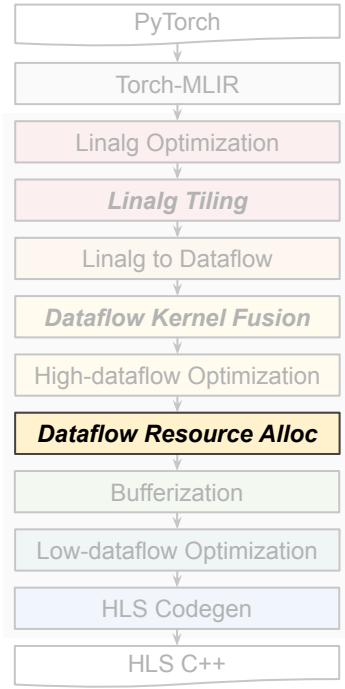
Source Token Production Curve (Push Imm. FIFO)

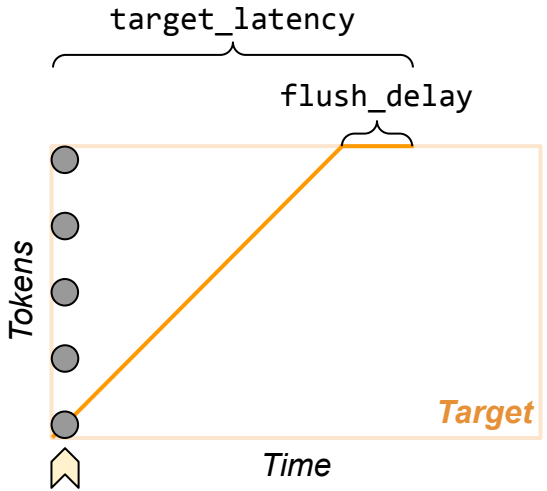
`token` = Atomic element passed between dataflow kernels

`longest_latency` = The longest latency among all the dataflow kernels in the accelerator, determining the maximum throughput



Token Number Estimation Heuristic (Cont.)

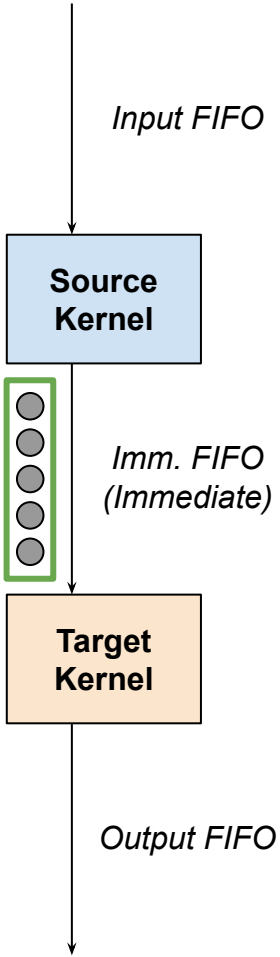




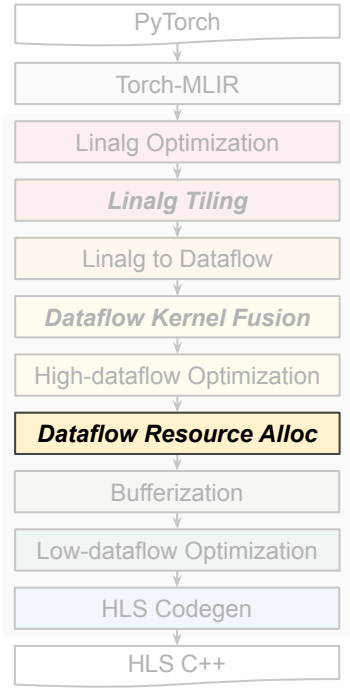
Target Token Consumption Curve (Pull Imm. FIFO)

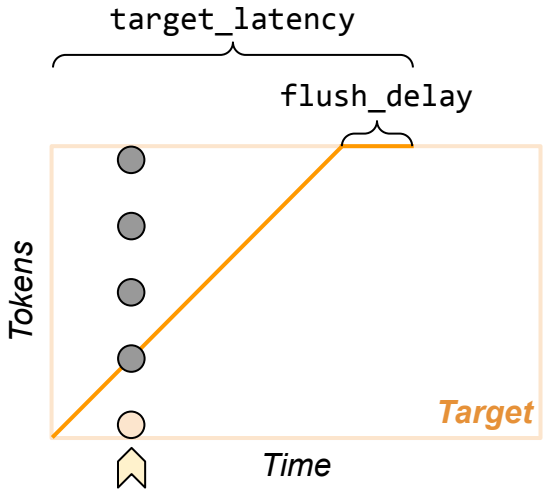
`token` = Atomic element passed between dataflow kernels

`longest_latency` = The longest latency among all the dataflow kernels in the accelerator, determining the maximum throughput



Token Number Estimation Heuristic (Cont.)

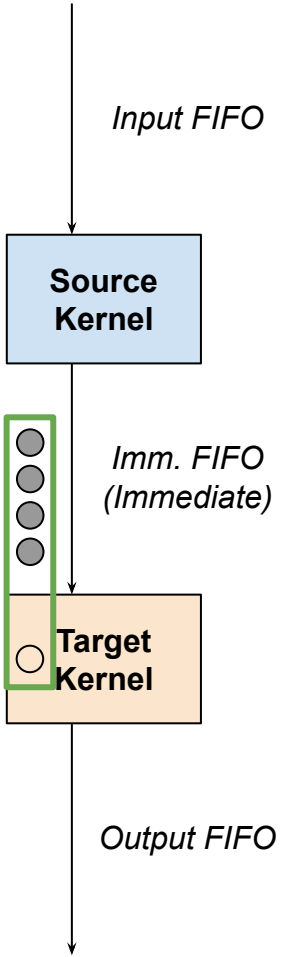




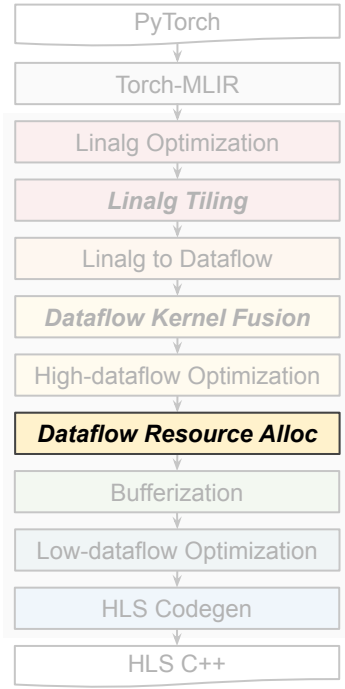
Target Token Consumption Curve (Pull Imm. FIFO)

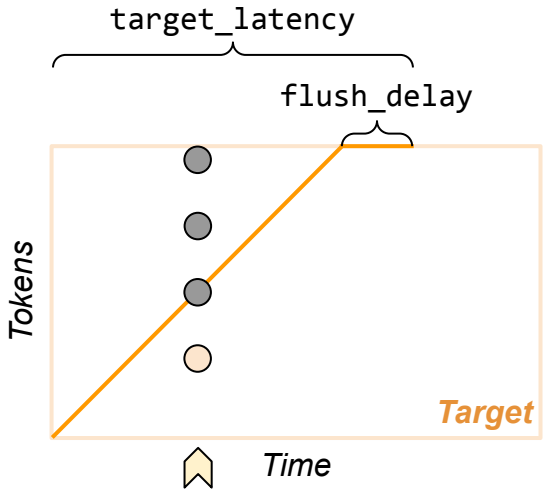
`token` = Atomic element passed between dataflow kernels

`longest_latency` = The longest latency among all the dataflow kernels in the accelerator, determining the maximum throughput



Token Number Estimation Heuristic (Cont.)

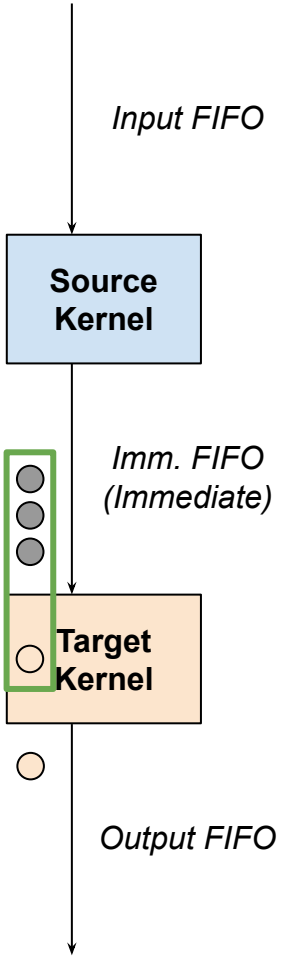




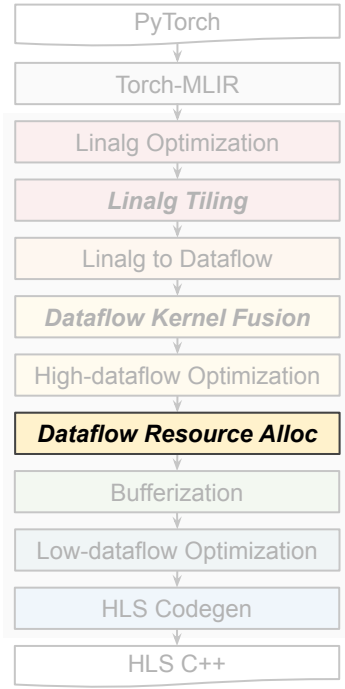
Target Token Consumption Curve (Pull Imm. FIFO)

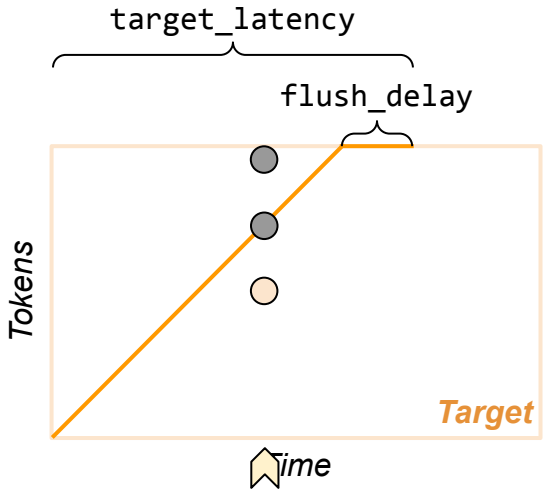
`token` = Atomic element passed between dataflow kernels

`longest_latency` = The longest latency among all the dataflow kernels in the accelerator, determining the maximum throughput



Token Number Estimation Heuristic (Cont.)

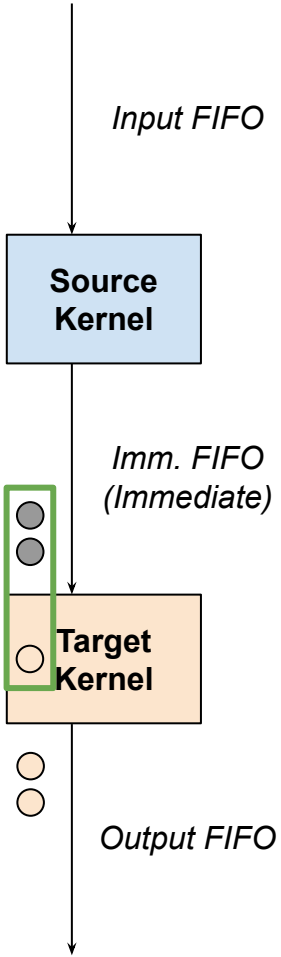




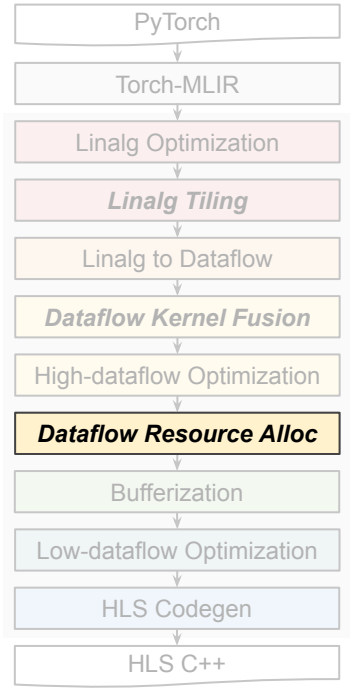
Target Token Consumption Curve (Pull Imm. FIFO)

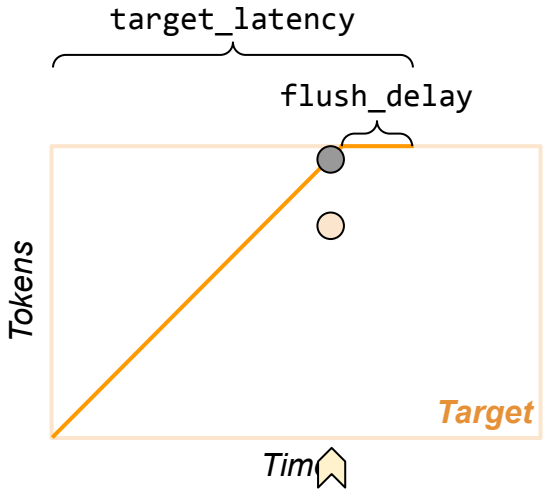
`token` = Atomic element passed between dataflow kernels

`longest_latency` = The longest latency among all the dataflow kernels in the accelerator, determining the maximum throughput



Token Number Estimation Heuristic (Cont.)

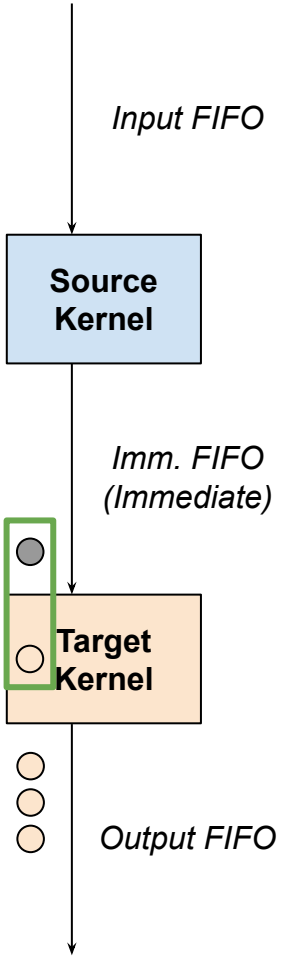




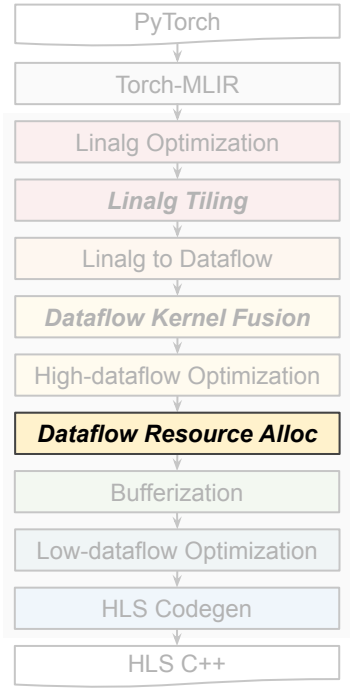
Target Token Consumption Curve (Pull Imm. FIFO)

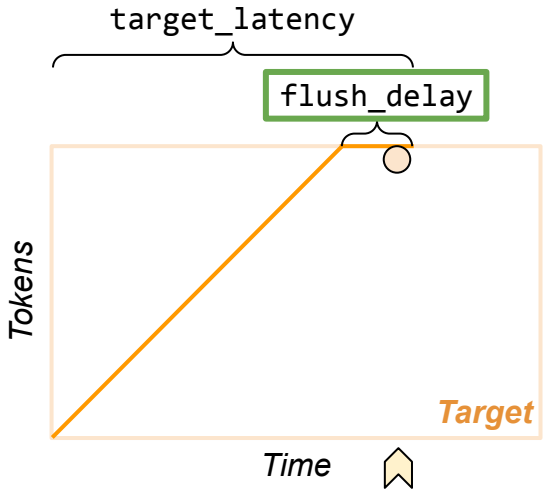
`token` = Atomic element passed between dataflow kernels

`longest_latency` = The longest latency among all the dataflow kernels in the accelerator, determining the maximum throughput



Token Number Estimation Heuristic (Cont.)

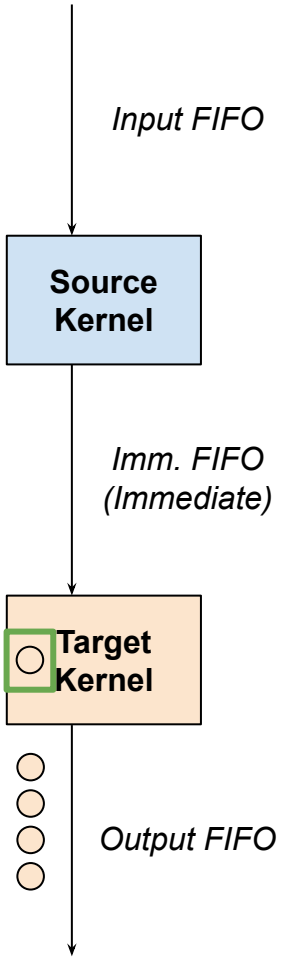




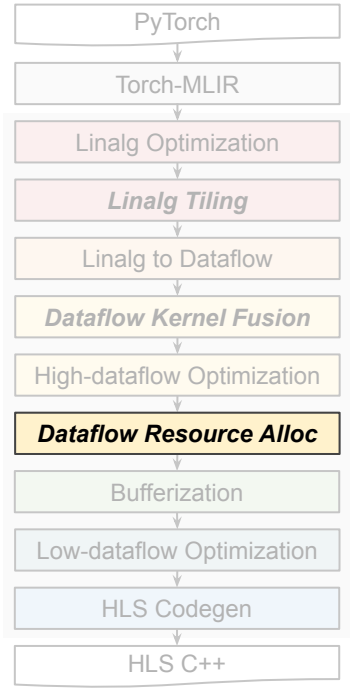
Target Token Consumption Curve (Pull Imm. FIFO)

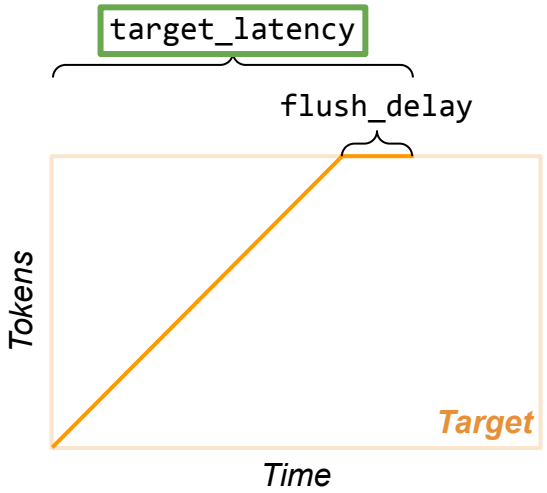
`token` = Atomic element passed between dataflow kernels

`longest_latency` = The longest latency among all the dataflow kernels in the accelerator, determining the maximum throughput



Token Number Estimation Heuristic (Cont.)

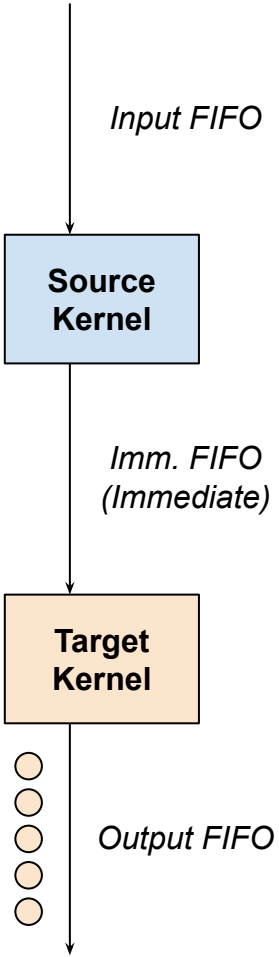




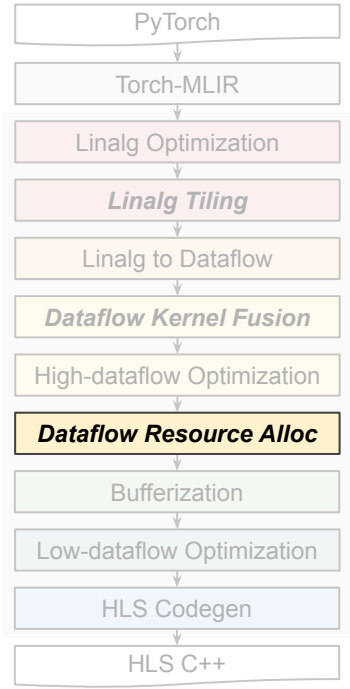
Target Token Consumption Curve (Pull Imm. FIFO)

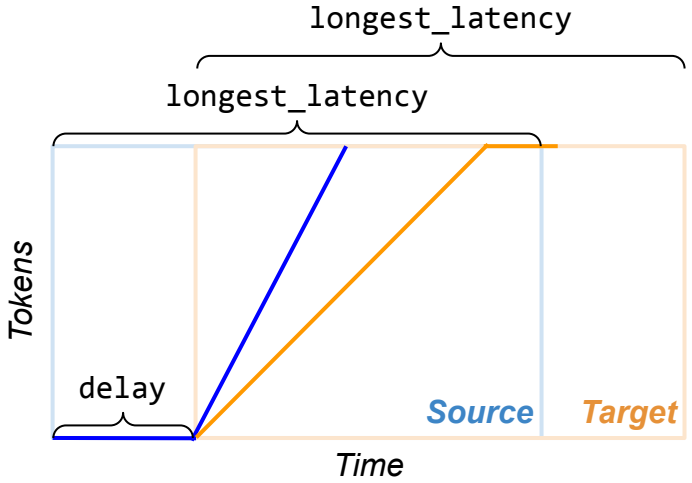
`token` = Atomic element passed between dataflow kernels

`longest_latency` = The longest latency among all the dataflow kernels in the accelerator, determining the maximum throughput



Token Number Estimation Heuristic (Cont.)

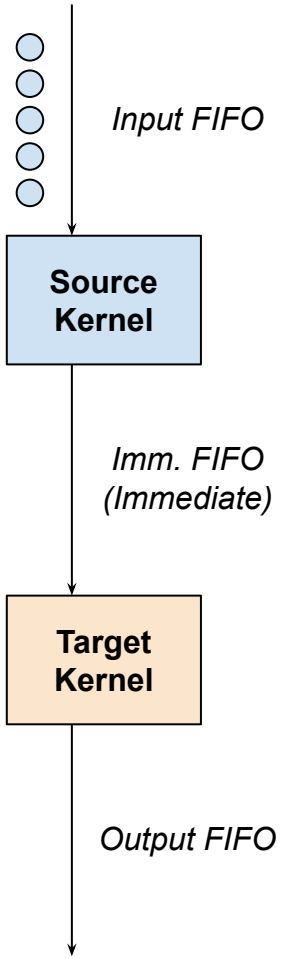




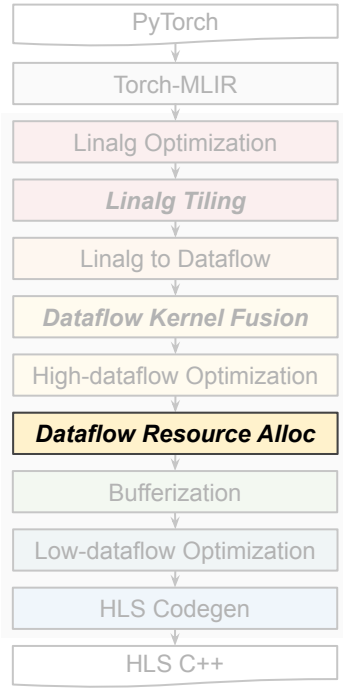
Overall Token Status Curve (Imm. FIFO)

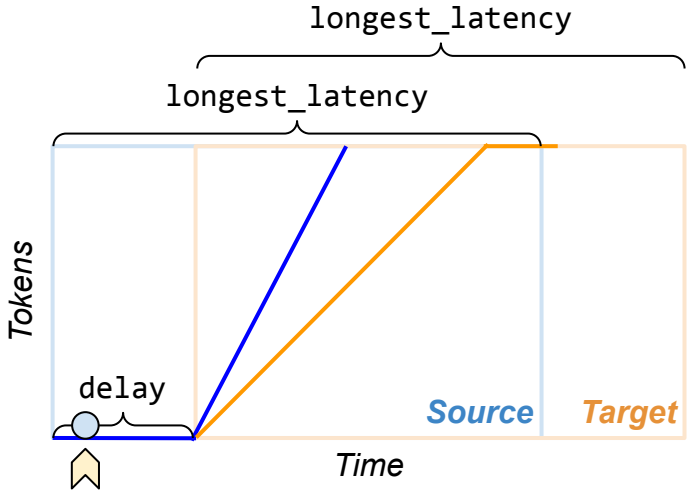
`token` = Atomic element passed between dataflow kernels

`longest_latency` = The longest latency among all the dataflow kernels in the accelerator, determining the maximum throughput



Token Number Estimation Heuristic (Cont.)

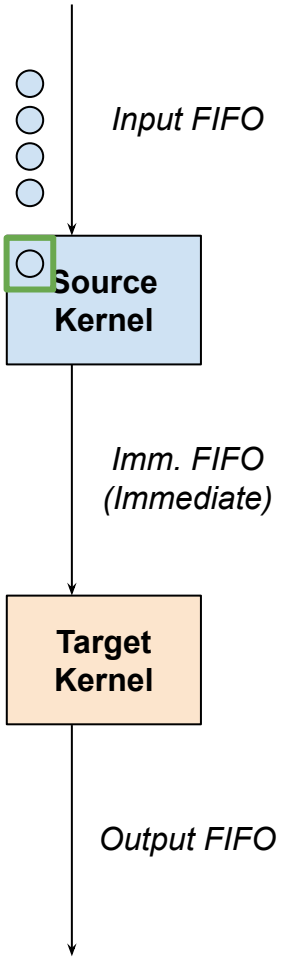




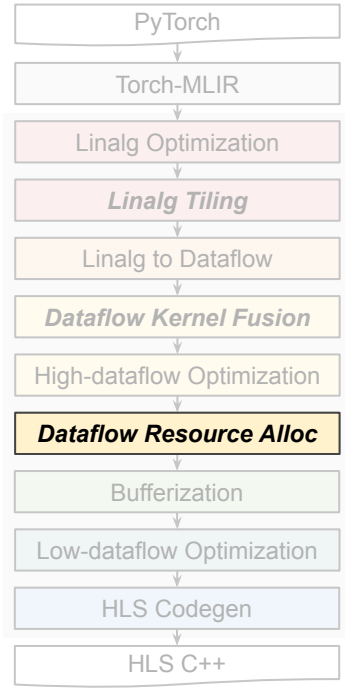
Overall Token Status Curve (Imm. FIFO)

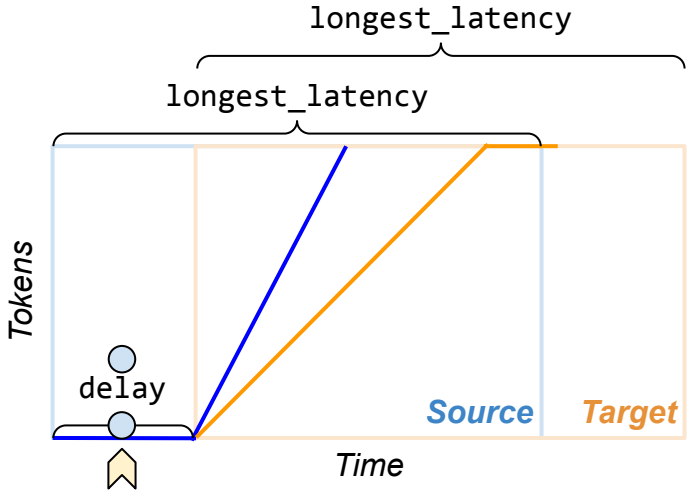
`token` = Atomic element passed between dataflow kernels

`longest_latency` = The longest latency among all the dataflow kernels in the accelerator, determining the maximum throughput



Token Number Estimation Heuristic (Cont.)

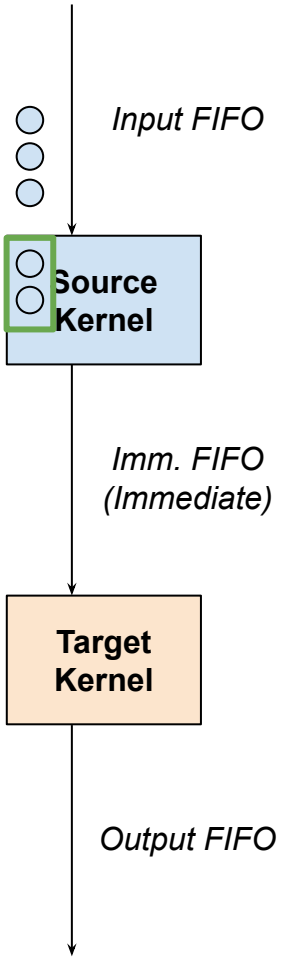




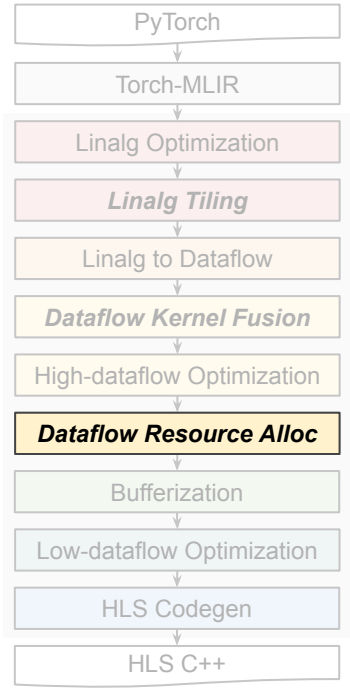
Overall Token Status Curve (Imm. FIFO)

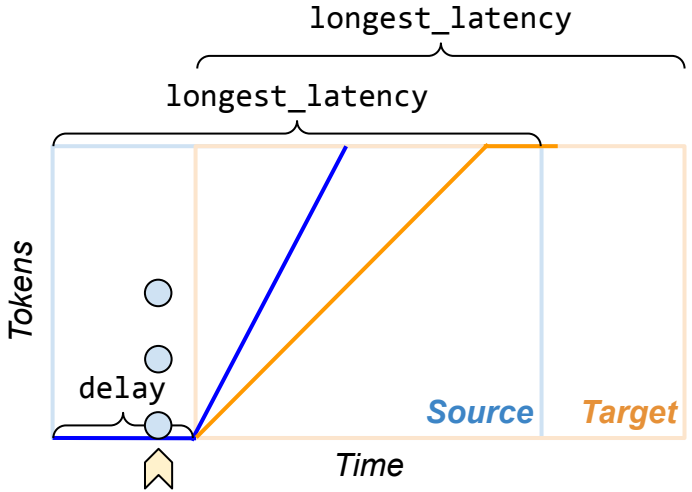
`token` = Atomic element passed between dataflow kernels

`longest_latency` = The longest latency among all the dataflow kernels in the accelerator, determining the maximum throughput



Token Number Estimation Heuristic (Cont.)

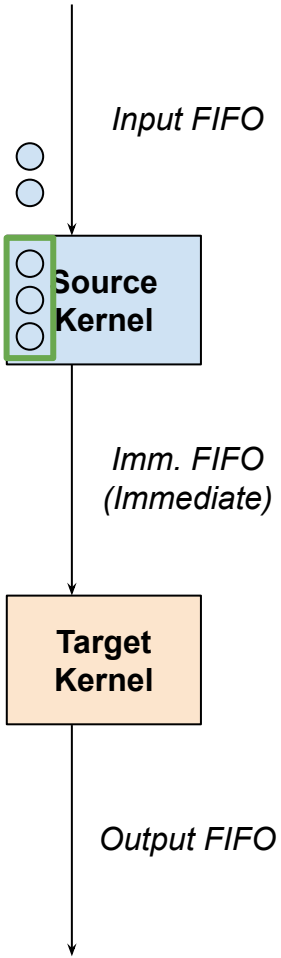




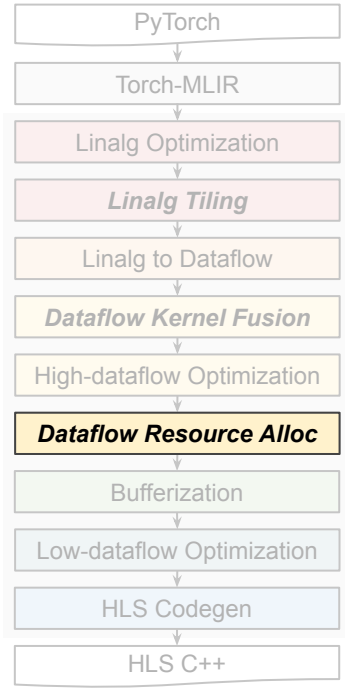
Overall Token Status Curve (Imm. FIFO)

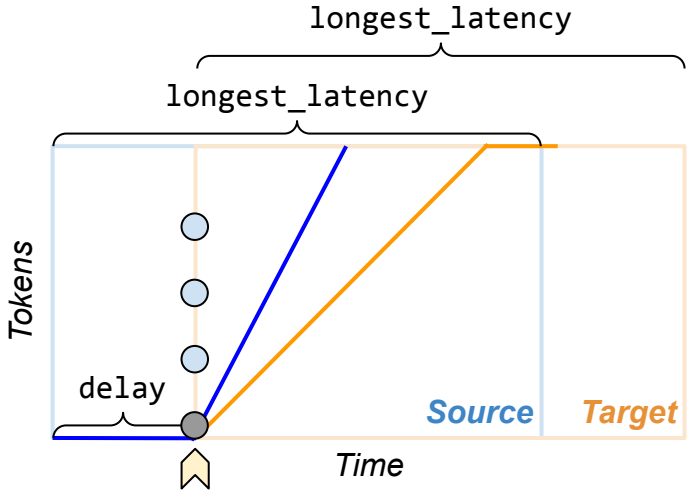
`token` = Atomic element passed between dataflow kernels

`longest_latency` = The longest latency among all the dataflow kernels in the accelerator, determining the maximum throughput



Token Number Estimation Heuristic (Cont.)

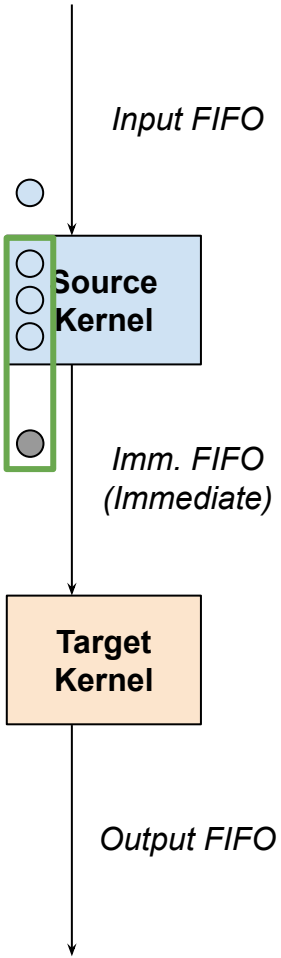




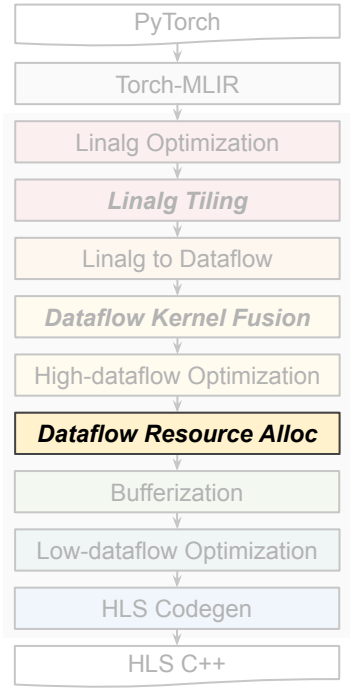
Overall Token Status Curve (Imm. FIFO)

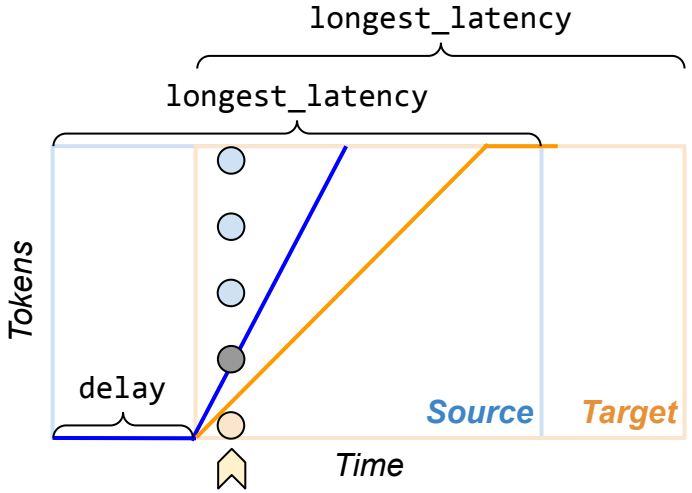
`token` = Atomic element passed between dataflow kernels

`longest_latency` = The longest latency among all the dataflow kernels in the accelerator, determining the maximum throughput



Token Number Estimation Heuristic (Cont.)

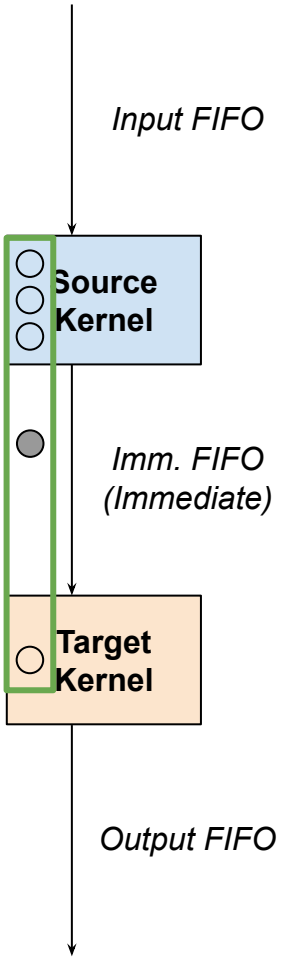




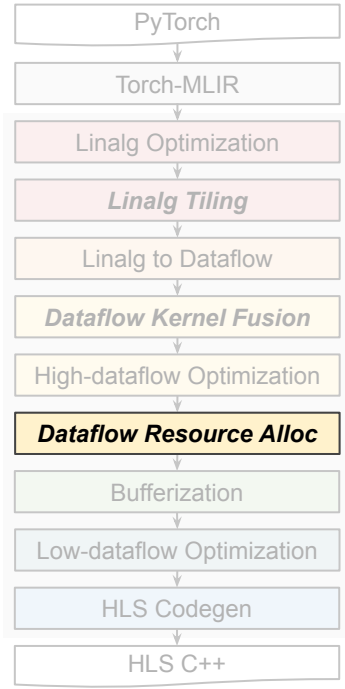
Overall Token Status Curve (Imm. FIFO)

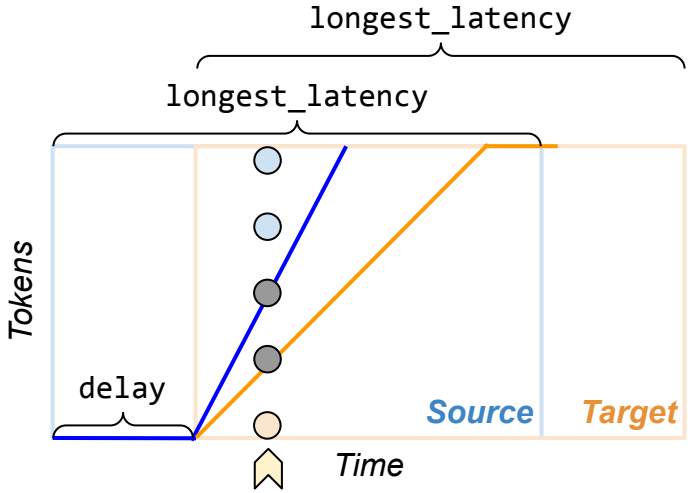
`token` = Atomic element passed between dataflow kernels

`longest_latency` = The longest latency among all the dataflow kernels in the accelerator, determining the maximum throughput



Token Number Estimation Heuristic (Cont.)

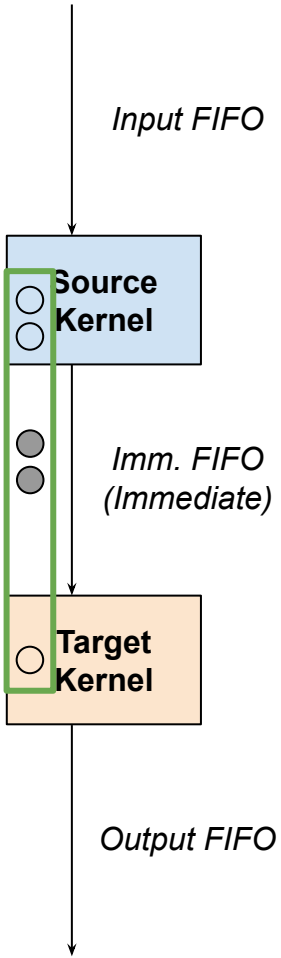




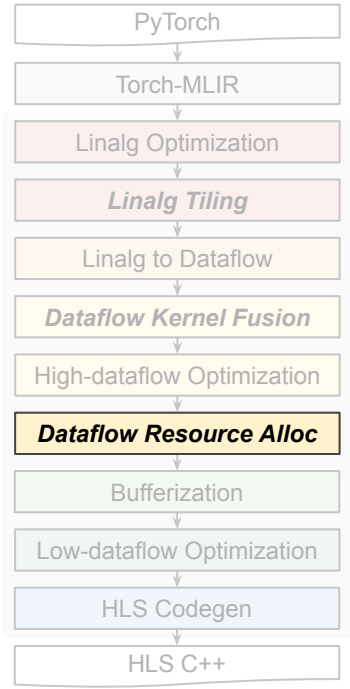
Overall Token Status Curve (Imm. FIFO)

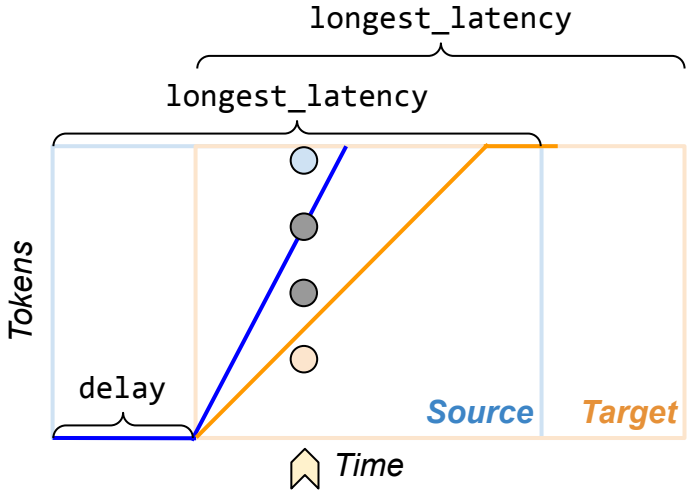
`token` = Atomic element passed between dataflow kernels

`longest_latency` = The longest latency among all the dataflow kernels in the accelerator, determining the maximum throughput



Token Number Estimation Heuristic (Cont.)

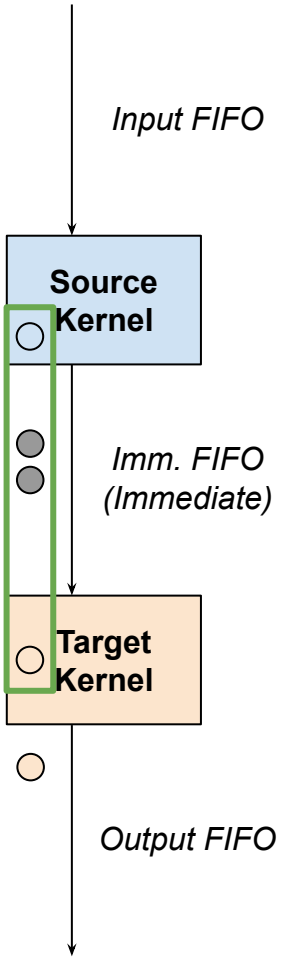




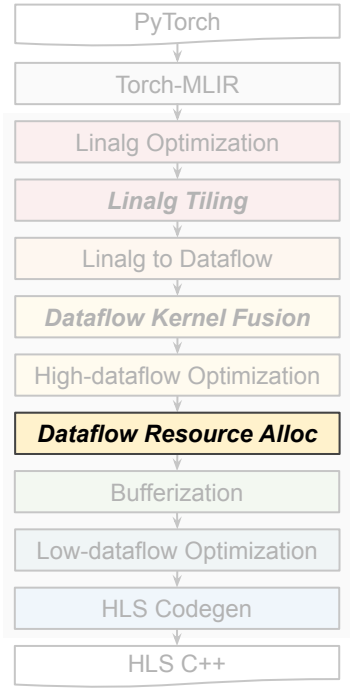
Overall Token Status Curve (Imm. FIFO)

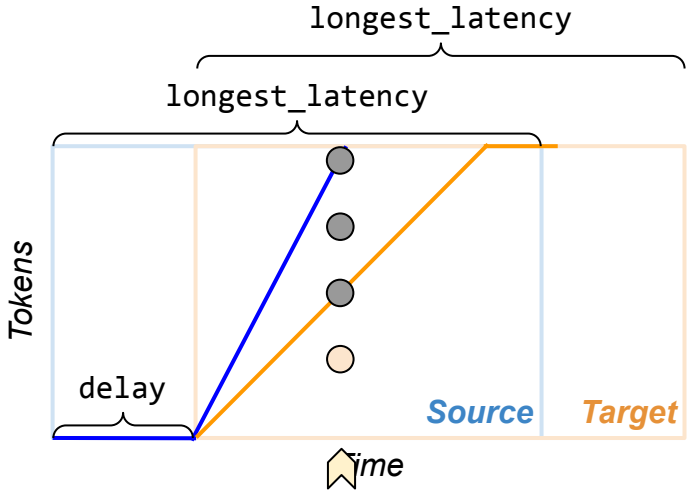
`token` = Atomic element passed between dataflow kernels

`longest_latency` = The longest latency among all the dataflow kernels in the accelerator, determining the maximum throughput



Token Number Estimation Heuristic (Cont.)

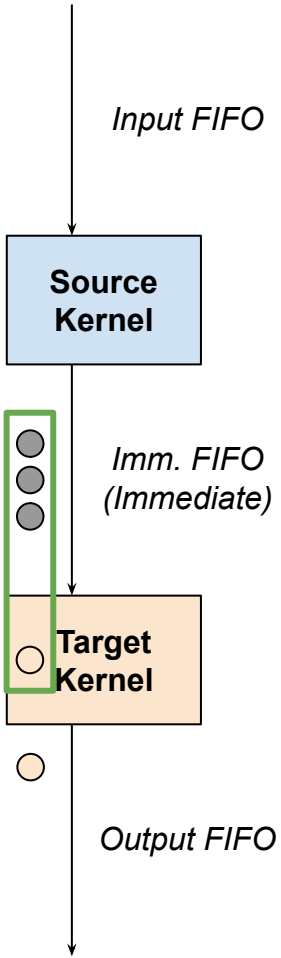




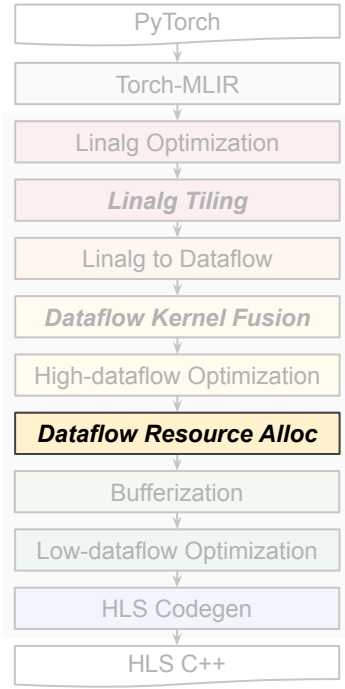
Overall Token Status Curve (Imm. FIFO)

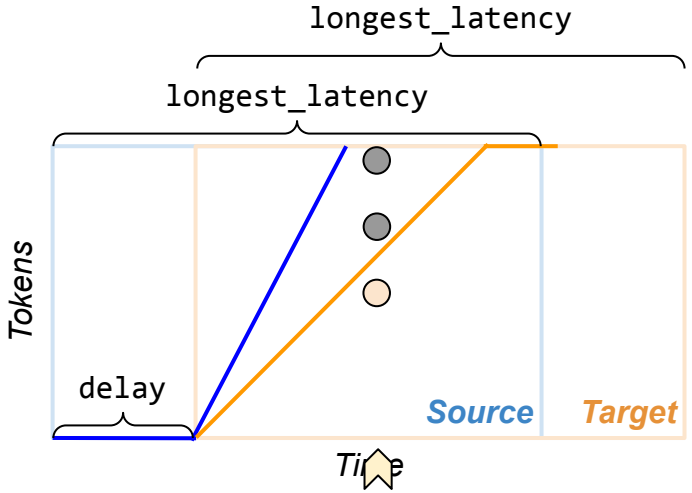
`token` = Atomic element passed between dataflow kernels

`longest_latency` = The longest latency among all the dataflow kernels in the accelerator, determining the maximum throughput



Token Number Estimation Heuristic (Cont.)

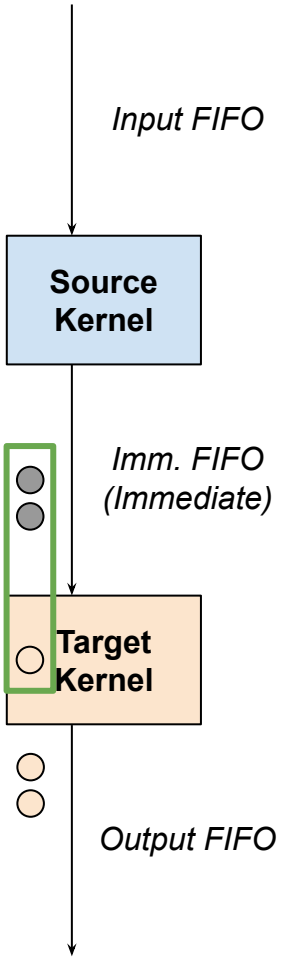




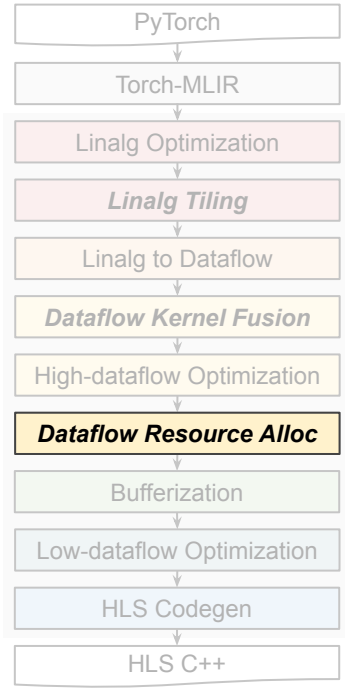
Overall Token Status Curve (Imm. FIFO)

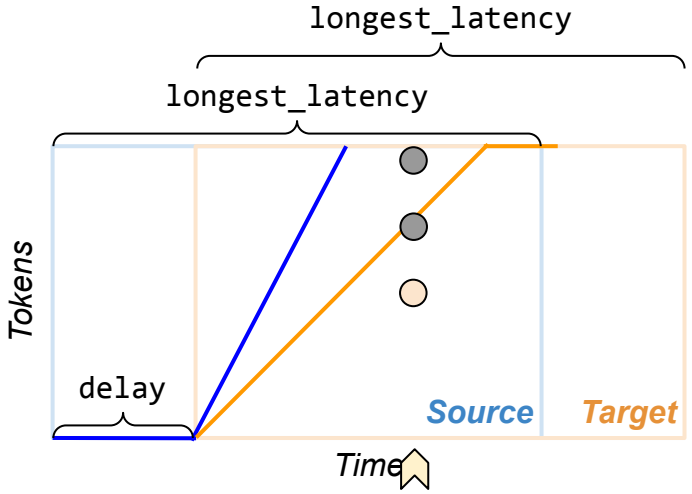
`token` = Atomic element passed between dataflow kernels

`longest_latency` = The longest latency among all the dataflow kernels in the accelerator, determining the maximum throughput



Token Number Estimation Heuristic (Cont.)

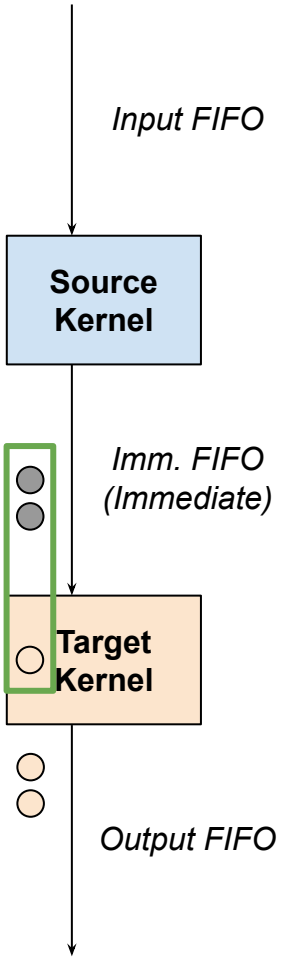




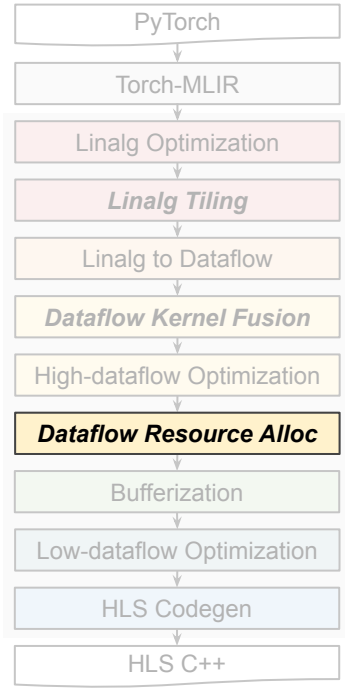
Overall Token Status Curve (Imm. FIFO)

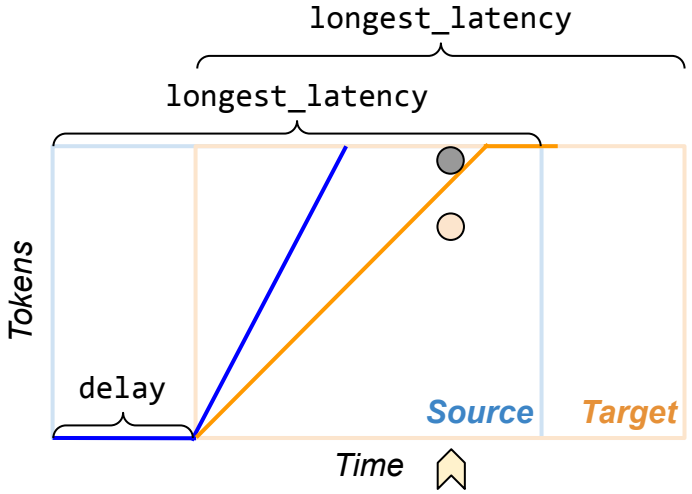
`token` = Atomic element passed between dataflow kernels

`longest_latency` = The longest latency among all the dataflow kernels in the accelerator, determining the maximum throughput



Token Number Estimation Heuristic (Cont.)

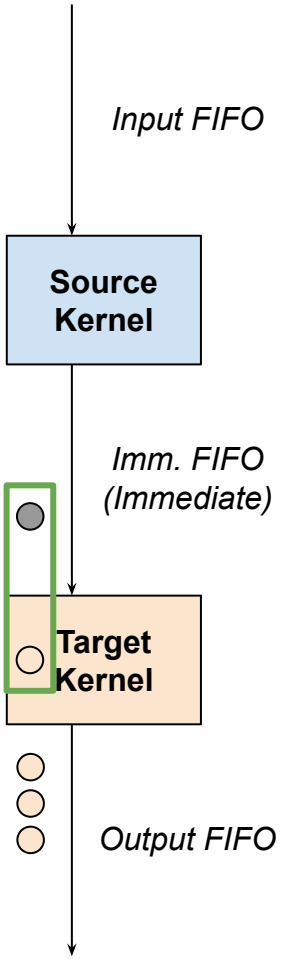




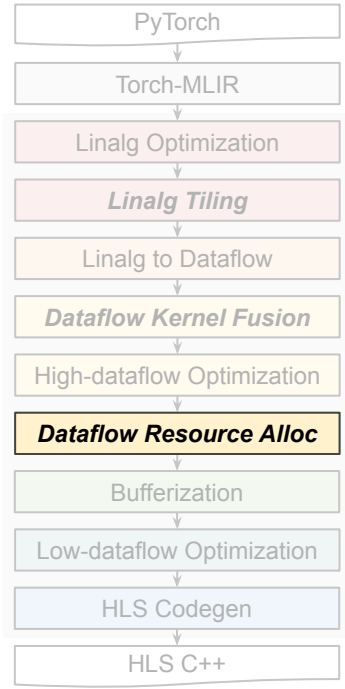
Overall Token Status Curve (Imm. FIFO)

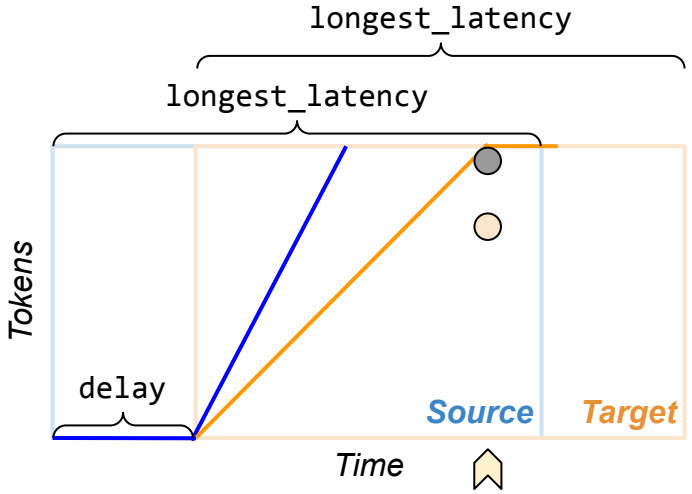
`token` = Atomic element passed between dataflow kernels

`longest_latency` = The longest latency among all the dataflow kernels in the accelerator, determining the maximum throughput



Token Number Estimation Heuristic (Cont.)

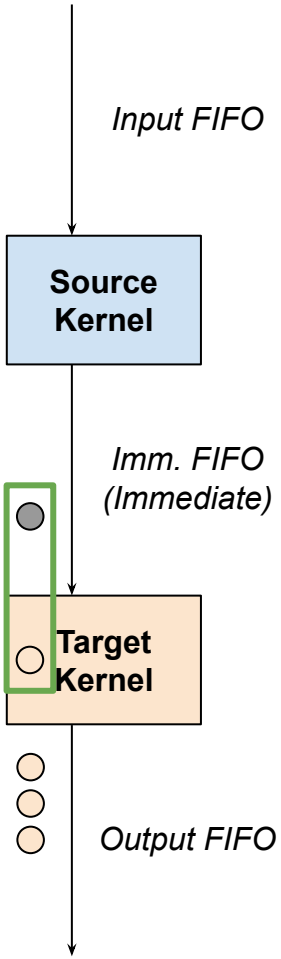




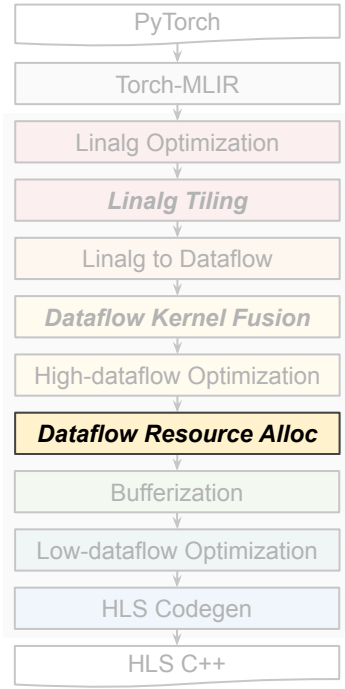
Overall Token Status Curve (Imm. FIFO)

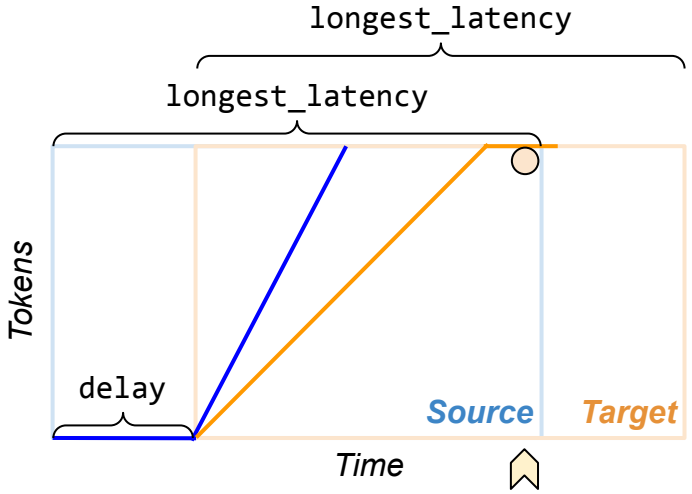
`token` = Atomic element passed between dataflow kernels

`longest_latency` = The longest latency among all the dataflow kernels in the accelerator, determining the maximum throughput



Token Number Estimation Heuristic (Cont.)

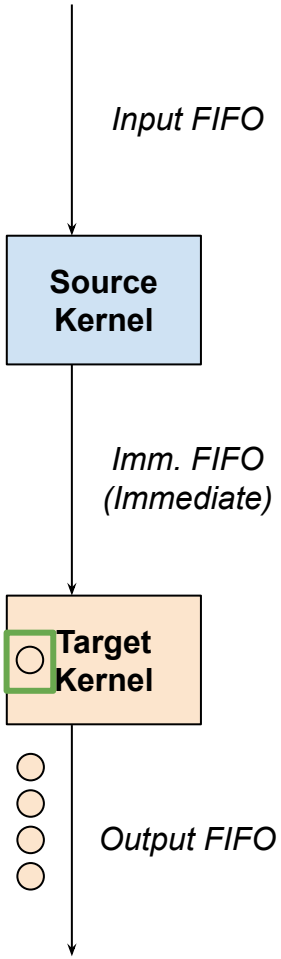




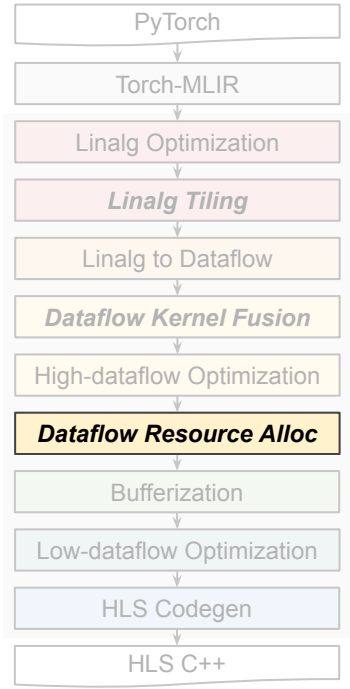
Overall Token Status Curve (Imm. FIFO)

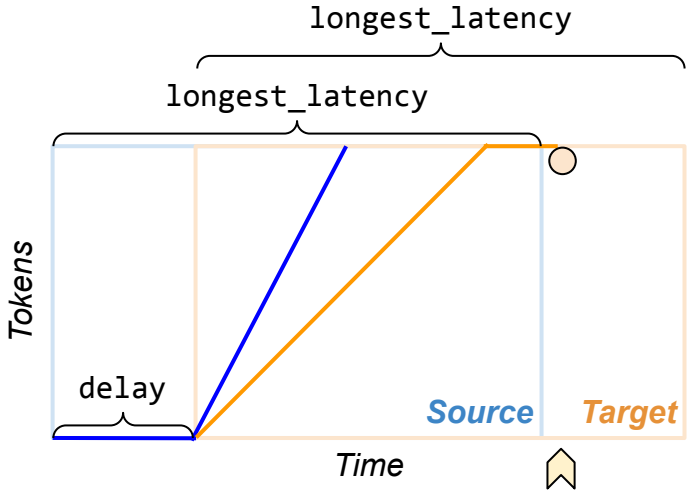
`token` = Atomic element passed between dataflow kernels

`longest_latency` = The longest latency among all the dataflow kernels in the accelerator, determining the maximum throughput



Token Number Estimation Heuristic (Cont.)

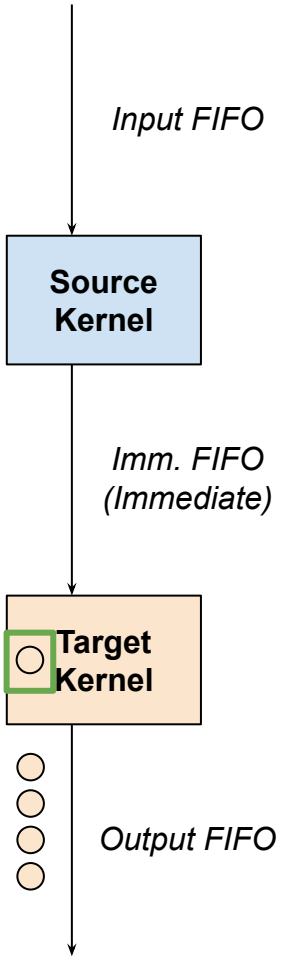




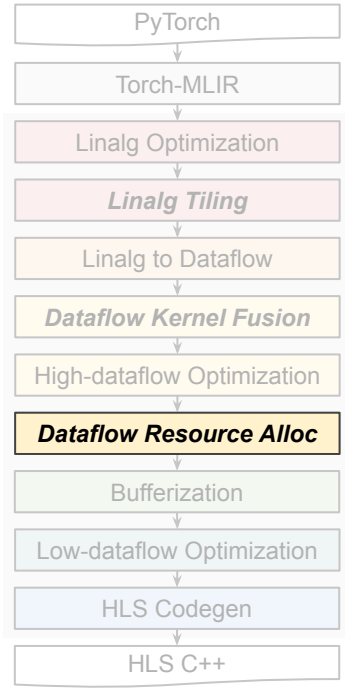
Overall Token Status Curve (Imm. FIFO)

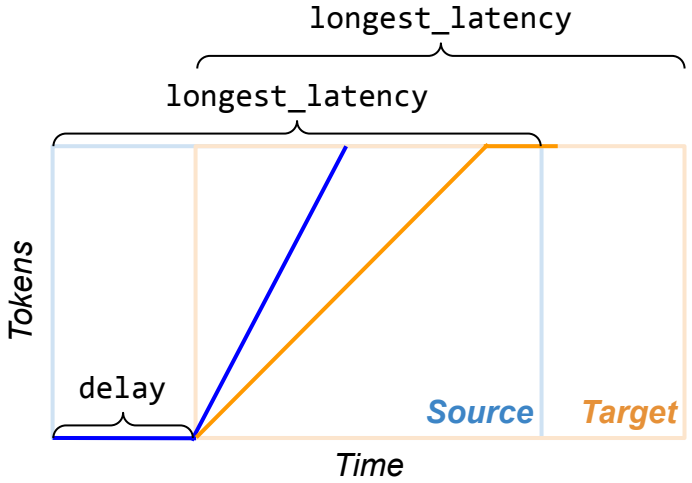
`token` = Atomic element passed between dataflow kernels

`longest_latency` = The longest latency among all the dataflow kernels in the accelerator, determining the maximum throughput



Token Number Estimation Heuristic (Cont.)

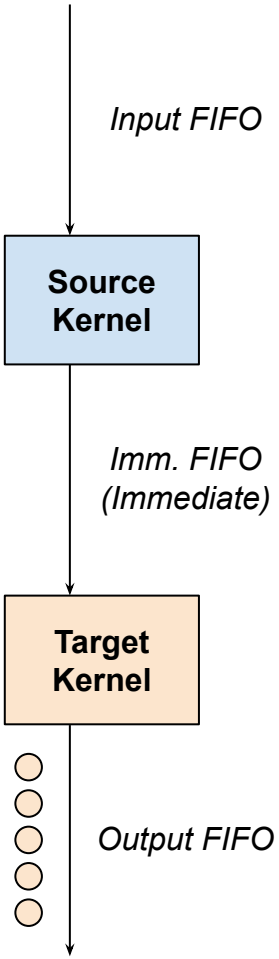




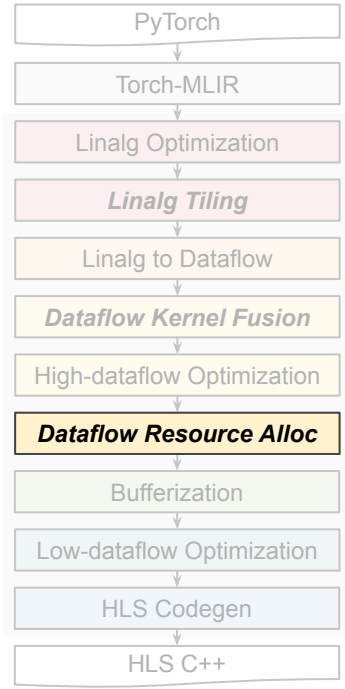
Overall Token Status Curve (Imm. FIFO)

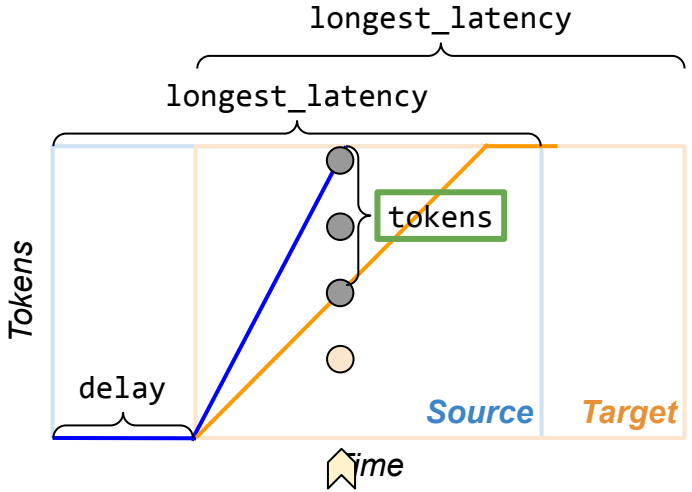
`token` = Atomic element passed between dataflow kernels

`longest_latency` = The longest latency among all the dataflow kernels in the accelerator, determining the maximum throughput



Token Number Estimation Heuristic (Cont.)

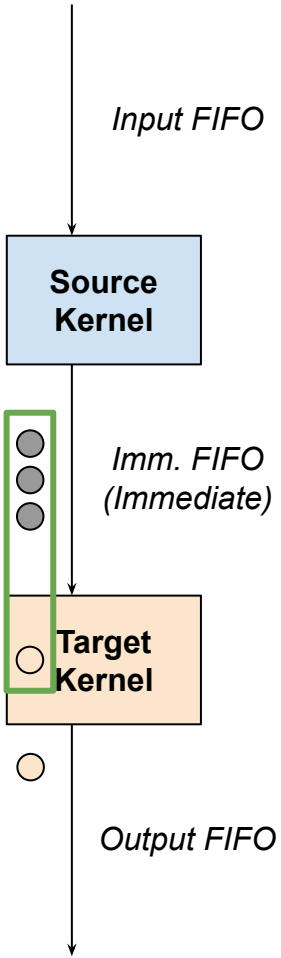




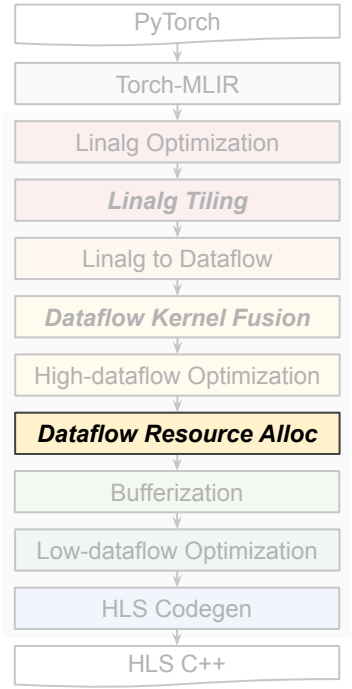
Overall Token Status Curve (Imm. FIFO)

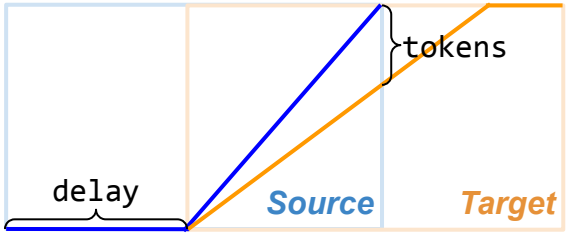
`token` = Atomic element passed between dataflow kernels

`longest_latency` = The longest latency among all the dataflow kernels in the accelerator, determining the maximum throughput



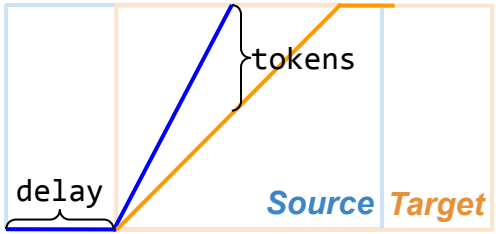
Token Number Estimation Strategies





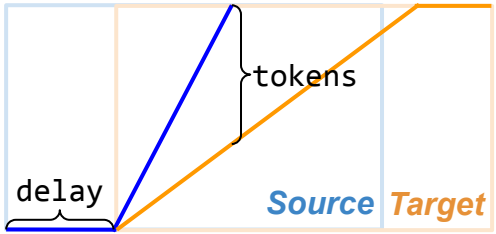
Conservative Strategy

Source Kernel: Slowest Speed = `longest_latency`
Target Kernel: Slowest Speed = `longest_latency`
delay: `init_delay*longest_latency/source_latency` (High ⤴️)
tokens: Low ⤵️



Normal Strategy

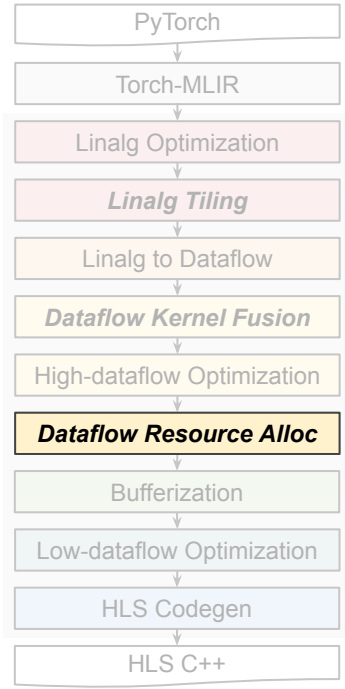
Source Kernel: Original Speed = `source_latency`
Target Kernel: Original Speed = `target_latency`
delay: `init_delay` (Low ⤵️)
tokens: Medium

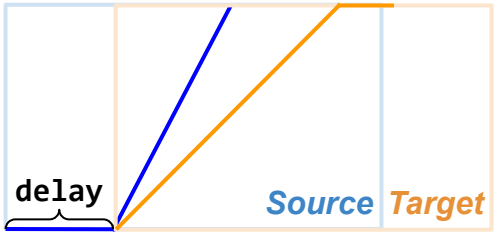


Aggressive Strategy

Source Kernel: Original Speed = `source_latency`
Target Kernel: Slowest Speed = `longest_latency`
delay: `init_delay` (Low ⤵️)
tokens: High ⤴️

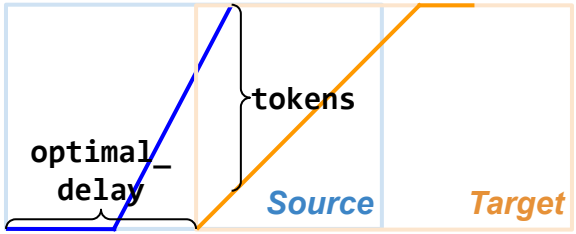
Linear Programming (LP) Formulation





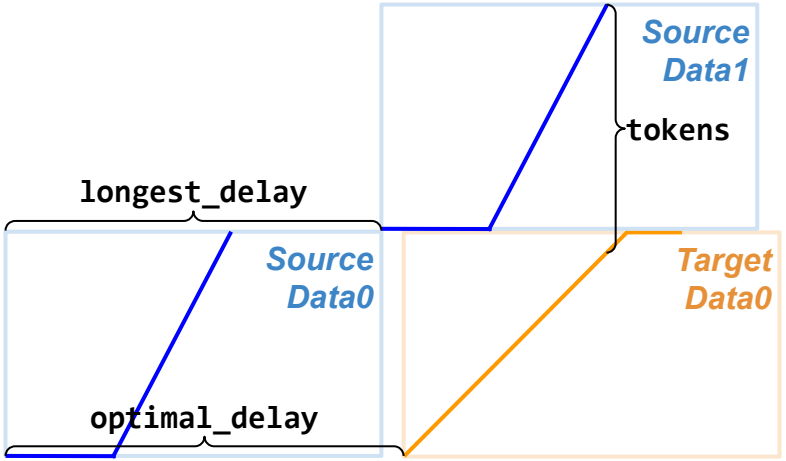
(1) Calculate token production/consumption curves based on kernel profiling results

(2) Calculate minimum `delay` given a strategy



(3) Solve `optimal_delay` with LP

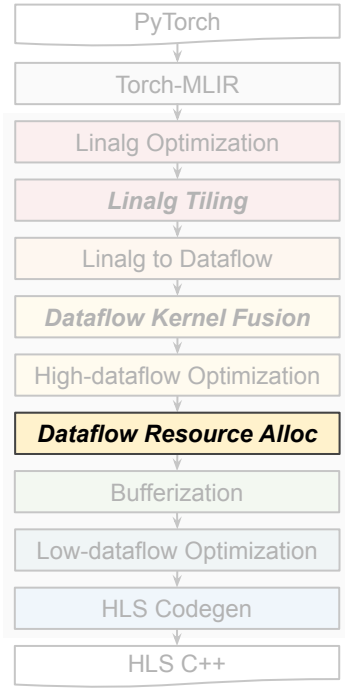
(4) Calculate `tokens` based on `optimal_delay` given a strategy

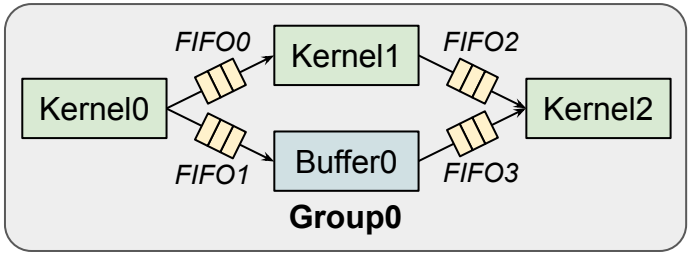


(5) Recalculate `tokens` if `optimal_delay` is larger than `longest_delay` while data-wise dataflow pipelining is enabled

Transform FIFO sizing into a scheduling problem

Linear Programming (LP) Formulation (Cont.)





Delay Variables

- `var["fifo0"]`
- `var["fifo1"]`
- `var["fifo2"]`
- `var["fifo3"]`

Minimum `delay` Constraints

- `var["fifo0"] >= delay["kernel0"]`
- `var["fifo1"] >= delay["kernel0"]`
- `var["fifo2"] >= delay["kernel1"]`
- `var["fifo3"] >= delay["buffer0"]`

Objective

- `obj = var["fifo0"] + var["fifo1"] + var["fifo2"] + var["fifo3"]`
- `minimize obj`

Path Balance Constraints

- `var["fifo0"] + var["fifo2"] >= critical_delay["kernel0"]["kernel2"]`
- `var["fifo1"] + var["fifo3"] >= critical_delay["kernel0"]["kernel2"]`

StreamTensor Outline

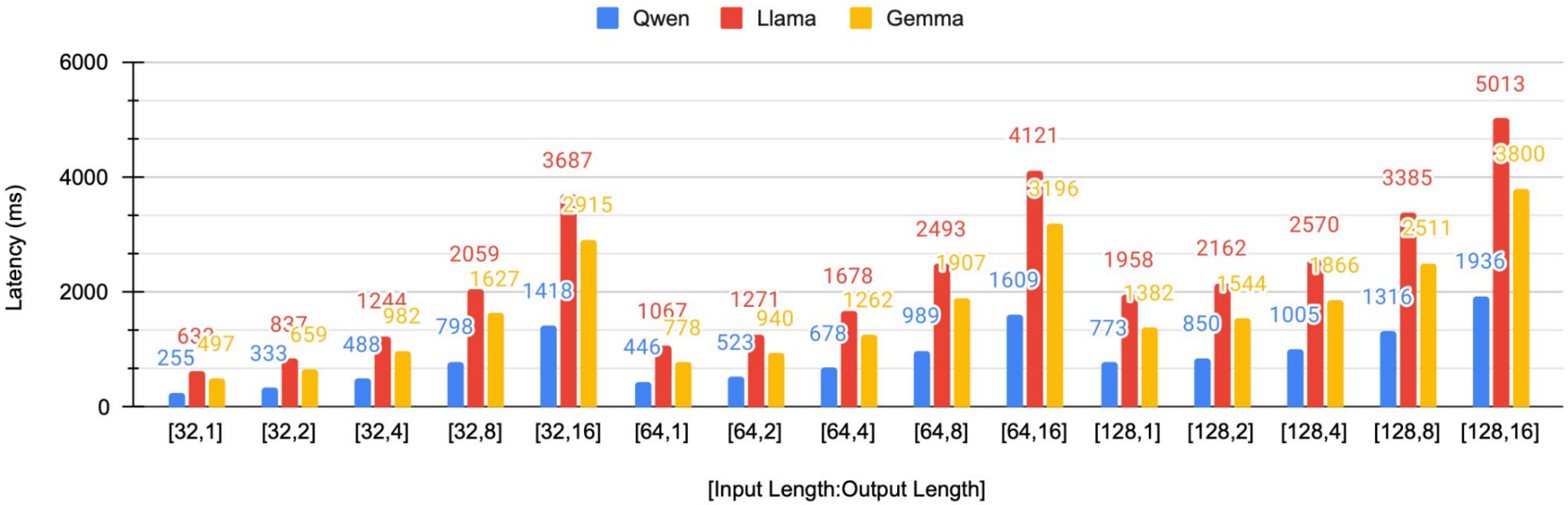
- Motivation
- StreamTensor Typing System
- StreamTensor Compilation Pipeline
- StreamTensor Design Spaces
- StreamTensor Results

FPGA On-board Results on GPT-2

[Input Len: Output Len]	Ours		Allo [8] (Ratio of $\frac{Ours}{Allo}$)		DFX [18] (Ratio of $\frac{Ours}{DFX}$)		A100 (Ratio of $\frac{Ours}{A100}$)		2080Ti (Ratio of $\frac{Ours}{2080Ti}$)	
	Latency (ms)	TTFT (ms)	Latency (ms)	TTFT (ms)	Latency (ms)	TTFT (ms)	Latency (ms)	TTFT (ms)	Latency (ms)	TTFT (ms)
[32:32]	194.99	34.59	238.32(0.82x)	81.50(0.42x)	350.00(0.56x)	177.20(0.20x)	291.16(0.67x)	8.72(3.97x)	518.46(0.38x)	24.98(1.38x)
[64:64]	358.24	61.27	476.64(0.75x)	162.99(0.38x)	694.70(0.52x)	349.10(0.18x)	567.41(0.63x)	8.76(6.99x)	1010.81(0.35x)	25.23(2.43x)
[128:128]	696.65	125.35	953.28(0.73x)	325.98(0.38x)	1384.00(0.50x)	692.80(0.18x)	1118.28(0.62x)	8.65(14.49x)	3969.76(0.18x)	25.26(4.96x)
[256:256]	1387.76	272.85	1906.56(0.73x)	651.96(0.42x)	2800.00(0.50x)	1417.60(0.19x)	2227.79(0.62x)	8.53(31.99x)	7914.23(0.18x)	25.23(10.81x)
Geo. Mean	-	-	**0.76x**	**0.40x**	**0.52x**	**0.19x**	**0.64x**	10.65x	**0.25x**	3.67x

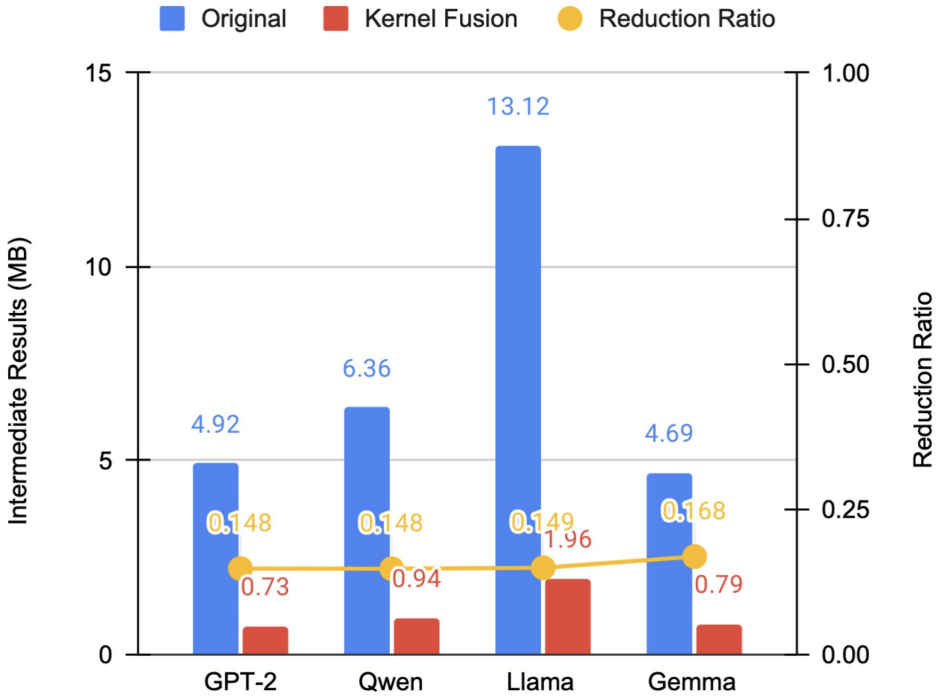
- StreamTensor outperforms previous FPGA LLM accelerators, Allo and DFX, on both overall latency and TTFT (time to first token).
- As an ADL, Allo follows the traditional dataflow accelerator design paradigm, demanding manual efforts for all the dataflow components design and comprehension of FIFO sizes, etc.
- StreamTensor outperforms NVIDIA GPU on overall latency, although performs considerably worse on TTFT, showing the advantages of dataflow accelerator on memory-bound application (LLM generation).

FPGA On-board Results on Emerging LLMs



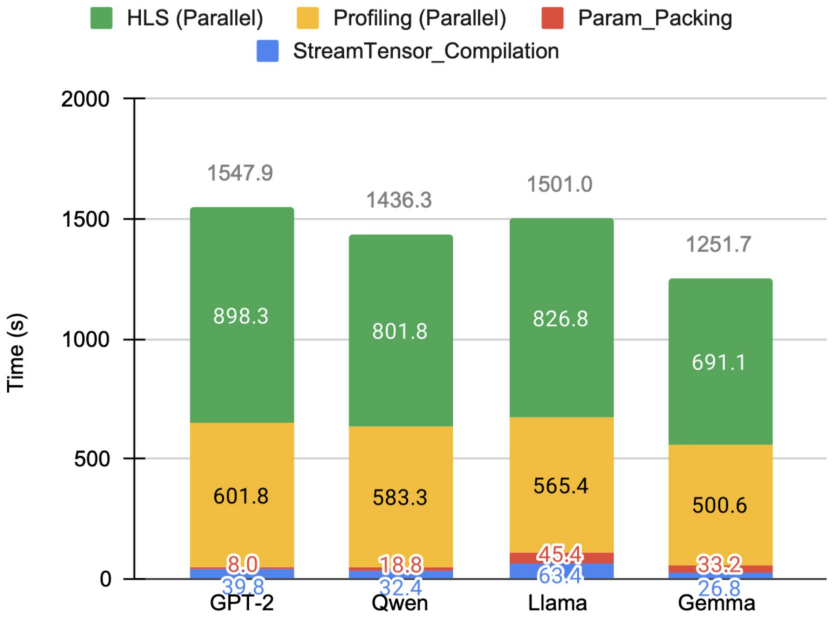
- Overall latency increases linearly as output length increases, showing a consistent decoding speed.
- TTFT (output length equal to 1) increases linearly as input length increases, because the MLP in LLM dominates the computation in our experiments.
- Note that GPT-2 is the only LLM reported in Allo and DFX due to their limited scalability and productivity.

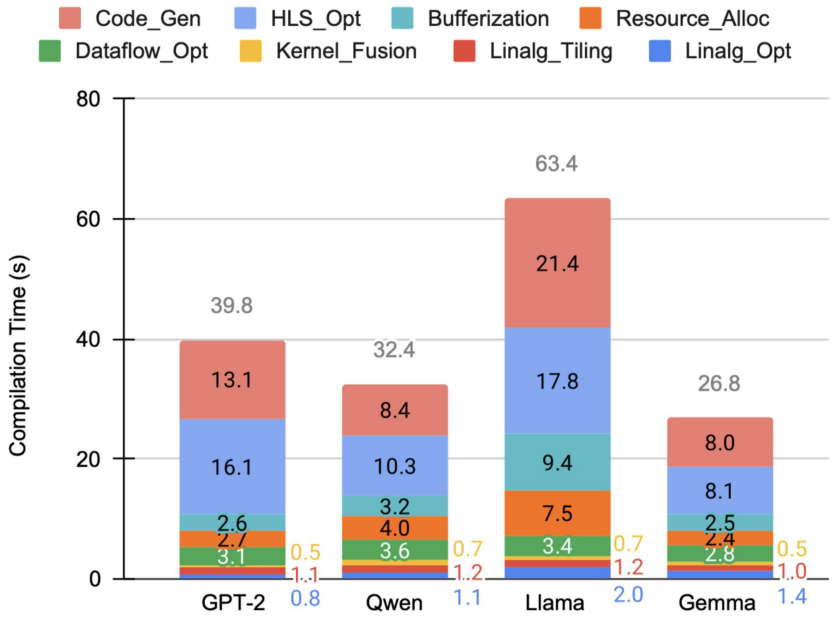
On-chip Memory Reduction through Kernel Fusion



- Only consider intermediate results, i.e., activations, in this study.
- Parameters are always stored in external memory.
- On-chip memory are reduced to 0.15x - 0.17x through stream-based kernel fusion.
- Without kernel fusion, the on-chip memory resources are not enough to store all intermediate results

RTL Generation Time





- For RTL generation, the downstream HLS synthesis and profiling consume most execution time
- StreamTensor compilation and parameter packing only consume a tiny portion of execution time
- For StreamTensor compilation, the low-level transforms (bufferization, HLS opt., and codegen) consume much more execution than high-level transforms (Linalg tiling, kernel fusion, dataflow opt.), showing the efficiency of the proposed itensor-based dataflow optimizations.

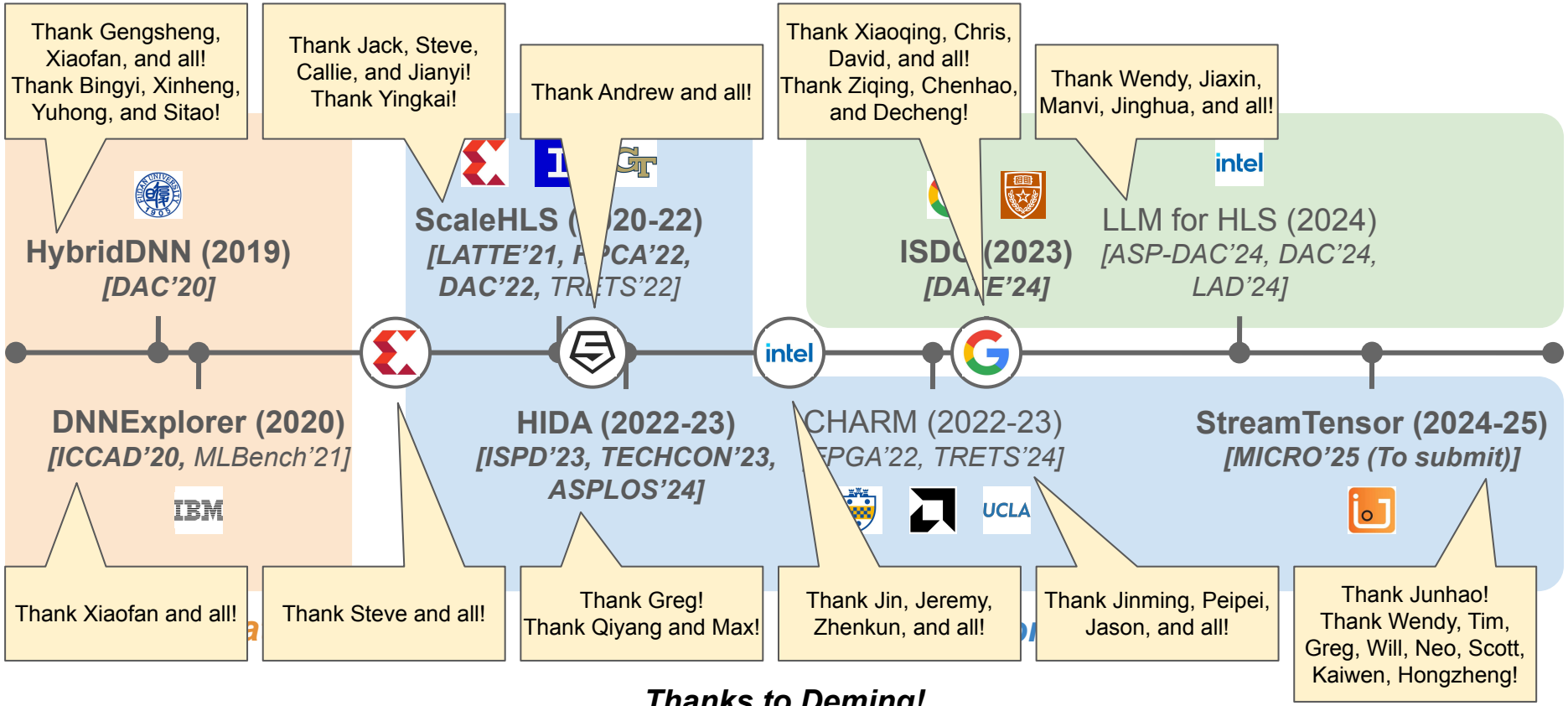
Conclusion and Acknowledgements

Conclusion

- We proposed ScaleHLS for single-kernel HLS optimizations:
  - Three levels of HLS representation and optimization, including graph, loop, and directive
  - Automated design space exploration for HLS loop pipelining, unrolling, and array partition.
- We proposed HIDA for multi-kernel HLS optimizations:
  - Two levels of dataflow representation, including functional and structural
  - Intensity and connectedness-aware design space exploration for multi-kernel designs
- We proposed StreamTensor for stream-based HLS optimizations:
  - Iterative tensor-based typing system for dataflow optimization and verification
  - Compilation pipeline for dataflow components generation, scheduling, and optimization
  - Three design spaces to improve dataflow efficiency, including linear algebra tiling, dataflow kernel fusion, and resource allocation

PyTorch-to-device Compilation based on HLS

Acknowledgements



Thanks to Deming!

Thanks to my parents, for their love and support.

Thank You!

Hanchen Ye, University of Illinois Urbana-Champaign
Advisor: Prof. Deming Chen
April 9, 2025

Final Exam Committee: Prof. Vikram Adve, Prof. Deming Chen (Chair), Prof. Jian Huang, Dr. Stephen Neuendorffer (Alphabetical Order)