Rethinking how we build compilers in a heterogeneous world

Michael O'Boyle
Senior EPSRC Research Fellow

Rethinking how we build compilers in a heterogeneous world

(or stealing ideas from other domains for our purposes)

Michael O'Boyle
Senior EPSRC Research Fellow

Rethinking how we build compilers in a heterogeneous world

(or stealing ideas from other domains for our purposes)

(or trying to make myself redundant with ML + endless automation)

Michael O'Boyle
Senior EPSRC Research Fellow

Rethinking how we build compilers in a heterogeneous world

Philip Ginsbach

Bruce Collie

Jackson Woodruff

Jordi Armengol Estape

Well known things

My view

Concrete results

Can we go further ?

Summary

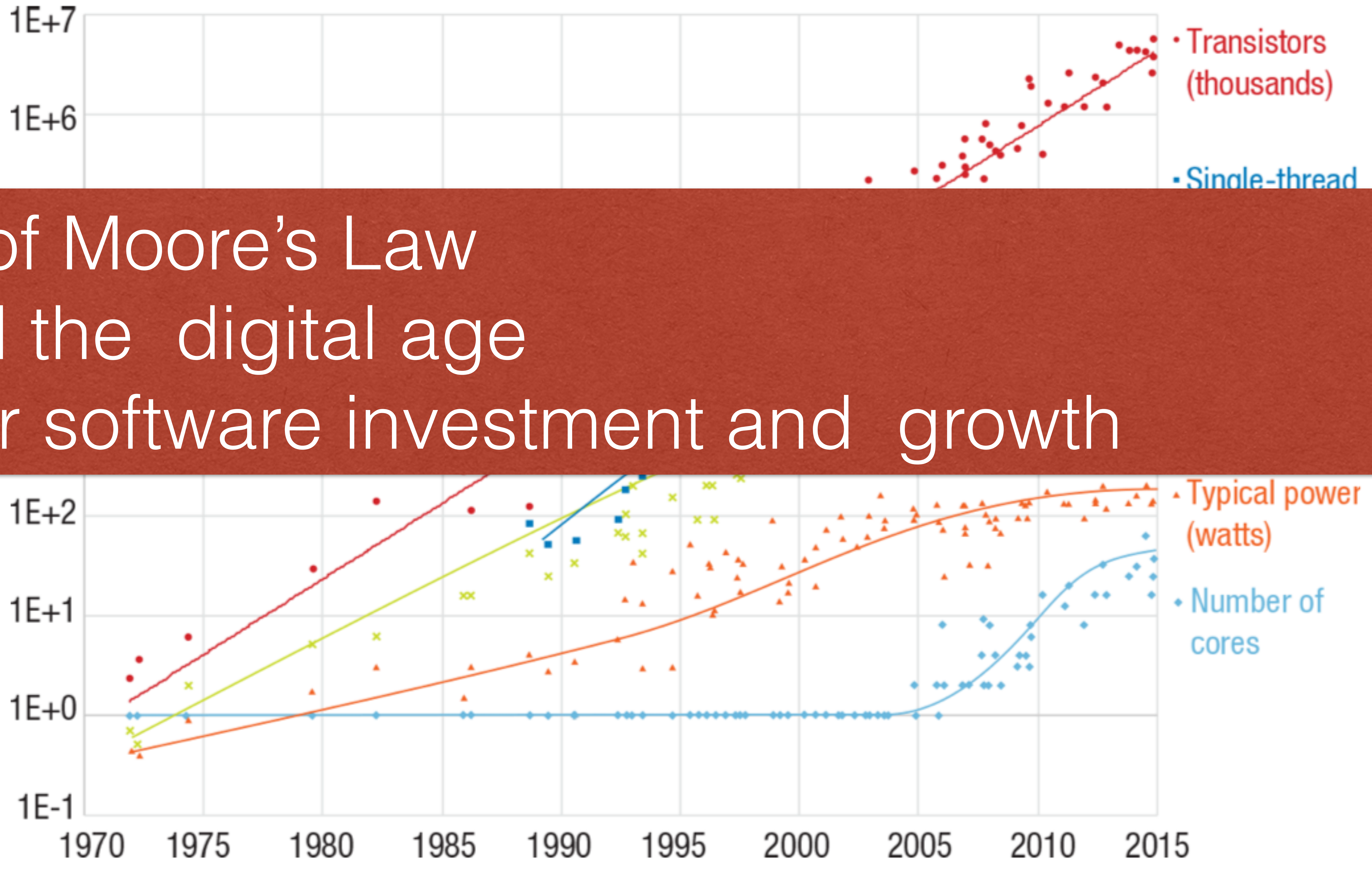
Well known things

My view

Concrete results

Can we go further ?

Summary

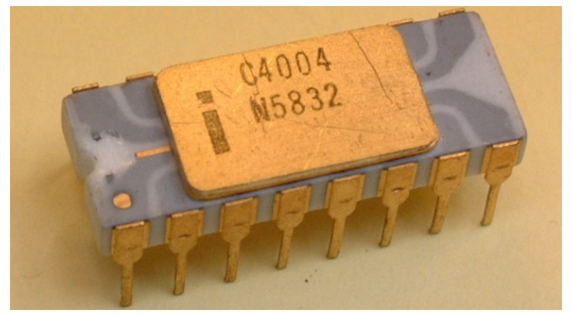


50 years of Moore's Law
- Enabled the digital age
- Basis for software investment and growth

Digital age based on a 50 year contract

Contract: Hardware may change "under the hood"

Hardware



Digital age based on a 50 year contract

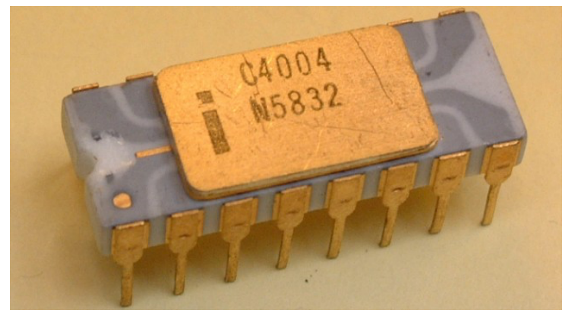
Contract: Hardware may change "under the hood"

BUT

Hardware/Software Interface remains constant

Software

Hardware



Digital age based on a 50 year contract

Contract: Hardware may change “under the hood”

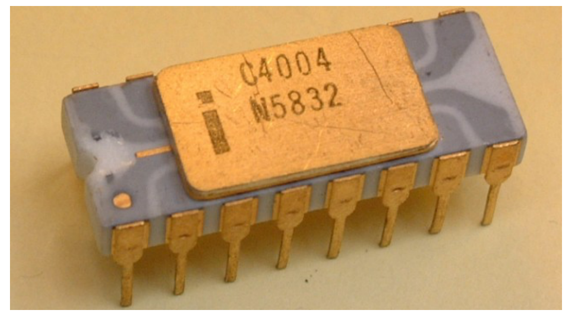
BUT

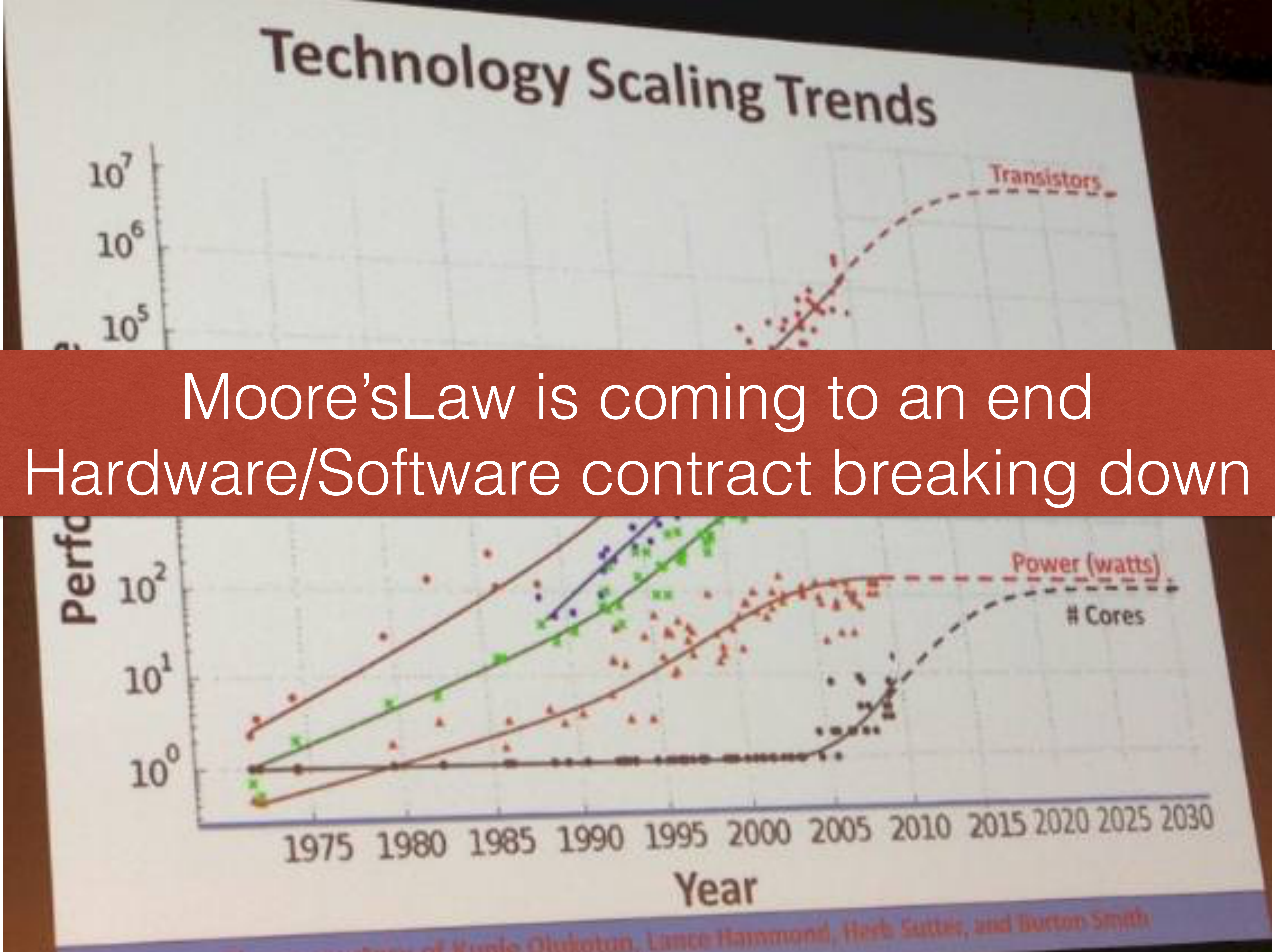
Hardware/Software Interface remains constant

Software written today guaranteed to run tomorrow

Software

Hardware

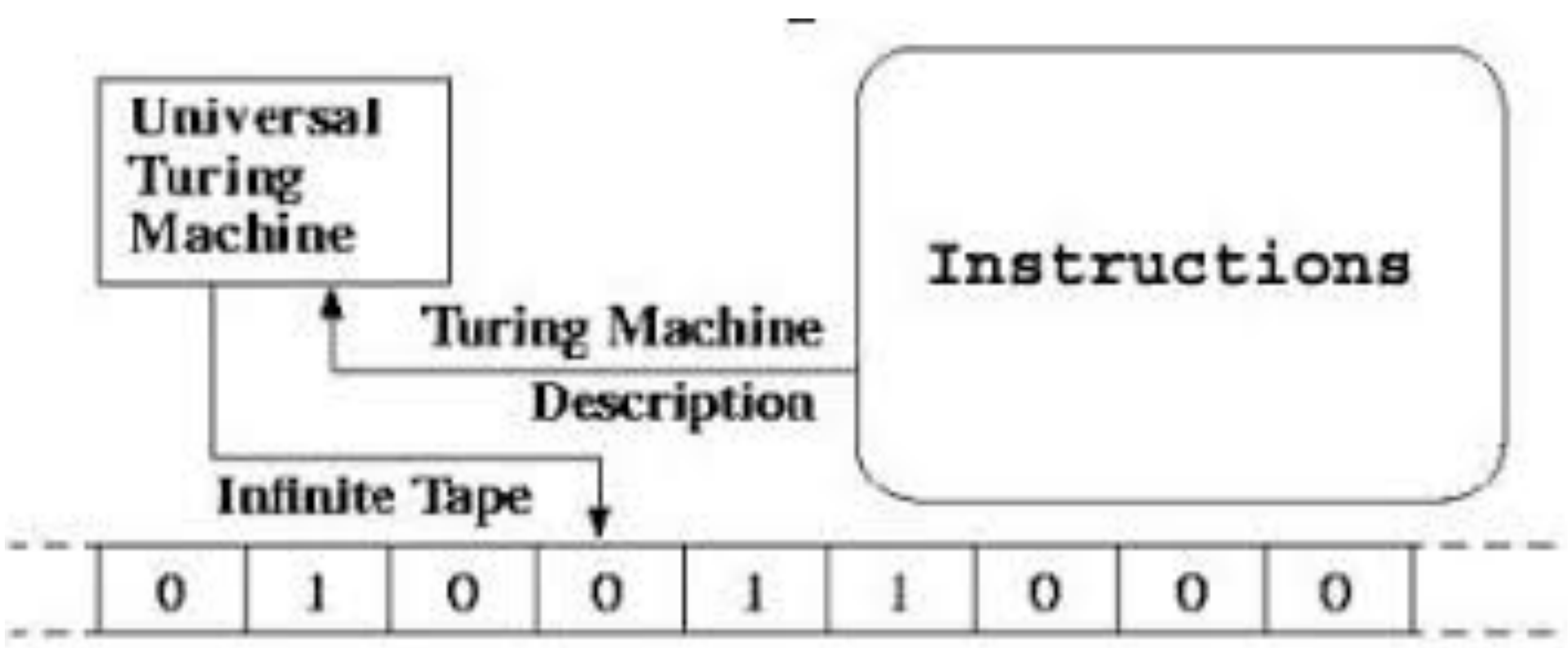


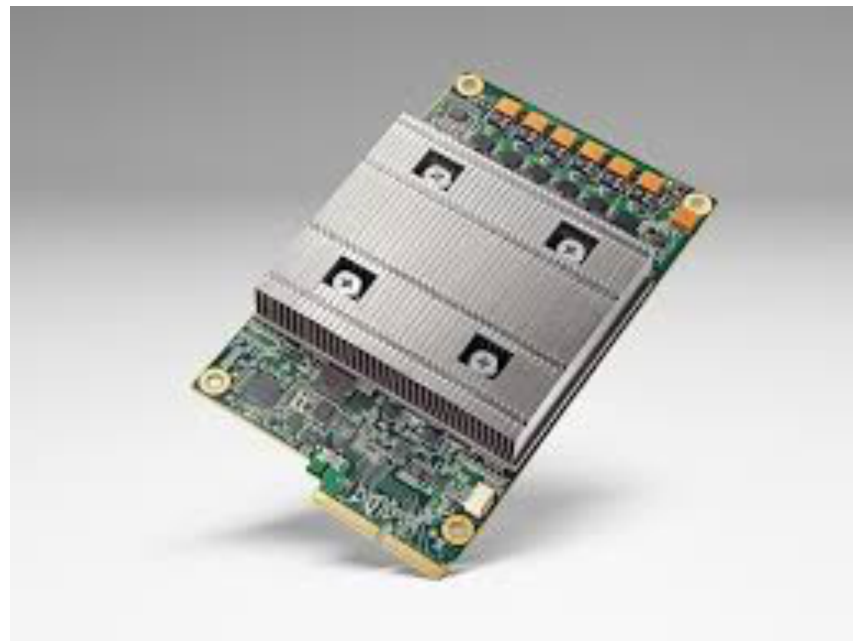


Moore'sLaw is coming to an end
Hardware/Software contract breaking down

Hardware/software contract breaking down

Technology trends means
- Hardware specialised or heterogenous

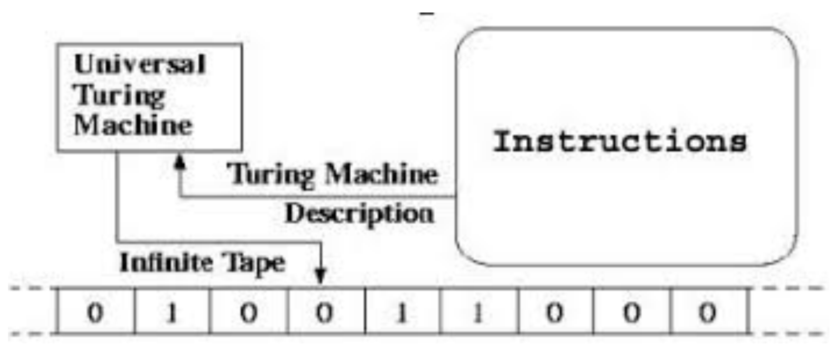




Hardware/software contract breaking down

Technology trends means
- Hardware specialised or heterogenous

Great
- up to 100,000x performance/energy gains



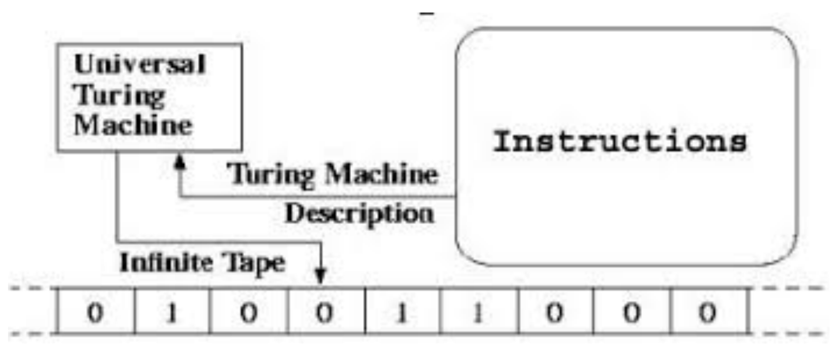
Hardware/software contract breaking down

Technology trends means
- Hardware specialised or heterogenous

Great
- up to 100,000x performance/energy gains

No free lunch
- Software cannot fit on new hardware

Heterogeneous crisis
- hardware stalls as software cannot fit



Hardware/software contract breaking down

Technology trends means
- Hardware specialised or heterogenous

Great

Rethink the contract

Heterogeneous crisis
- hardware stalls as software cannot fit

Not the first person to notice this

Well known things

My view

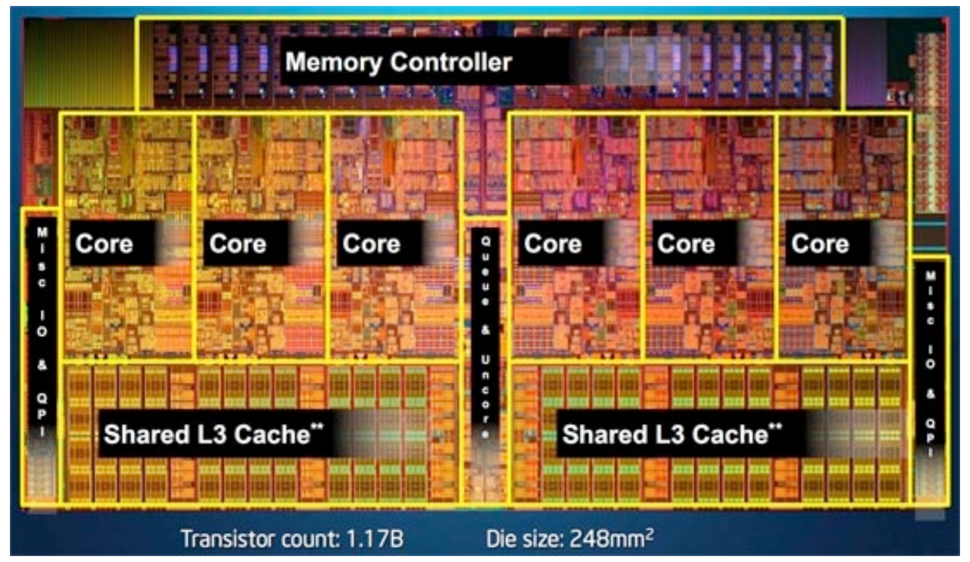
Concrete results

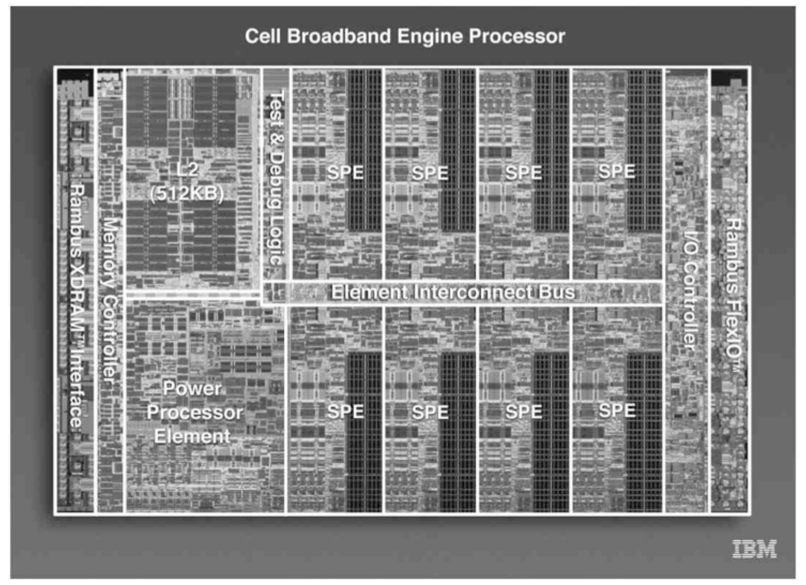
Can we go further ?

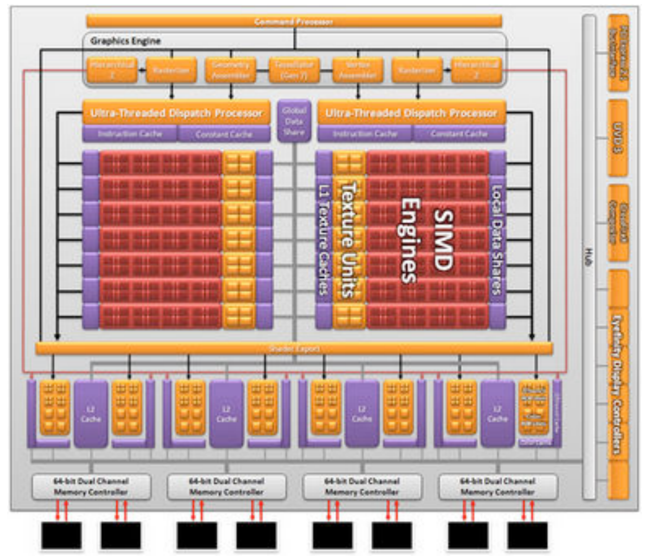
Summary

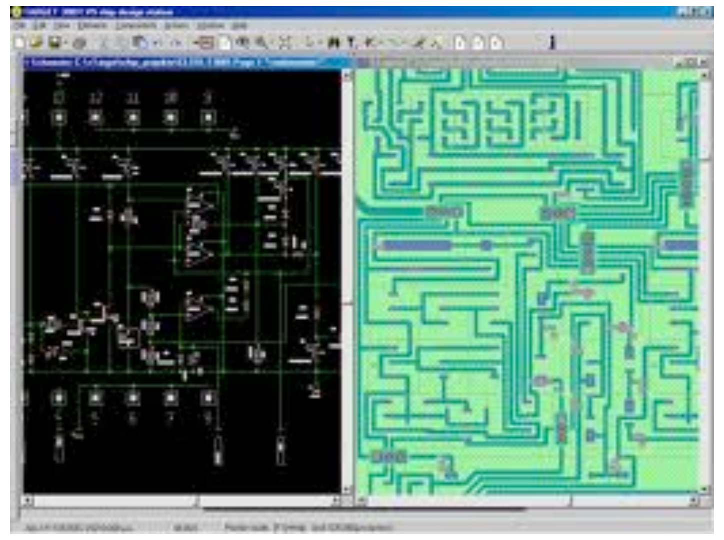
How to bridge the gap?

New Application/Legacy Code





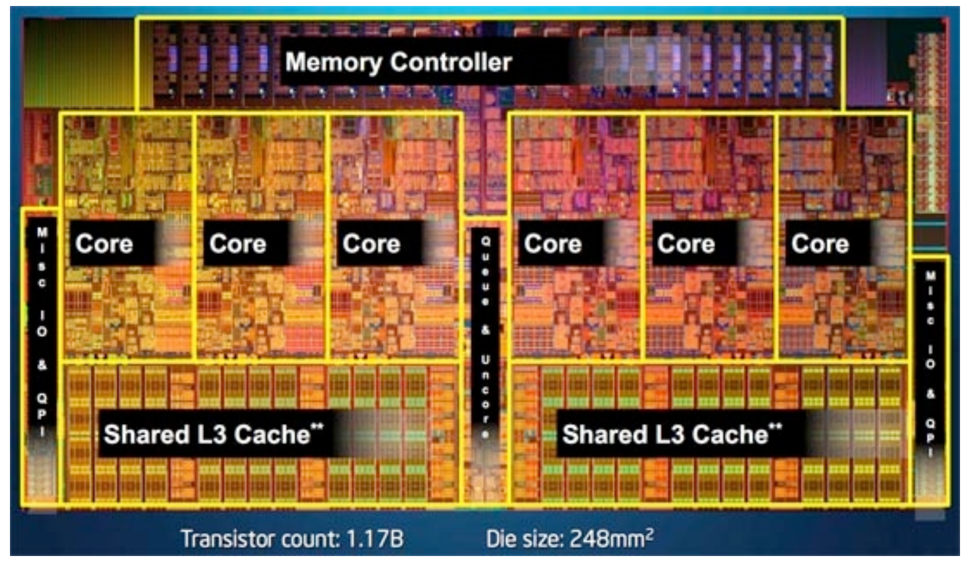


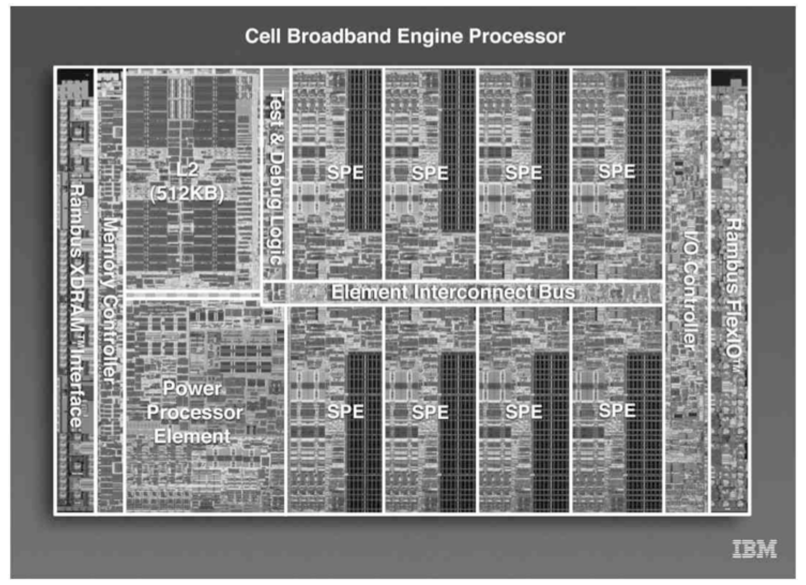


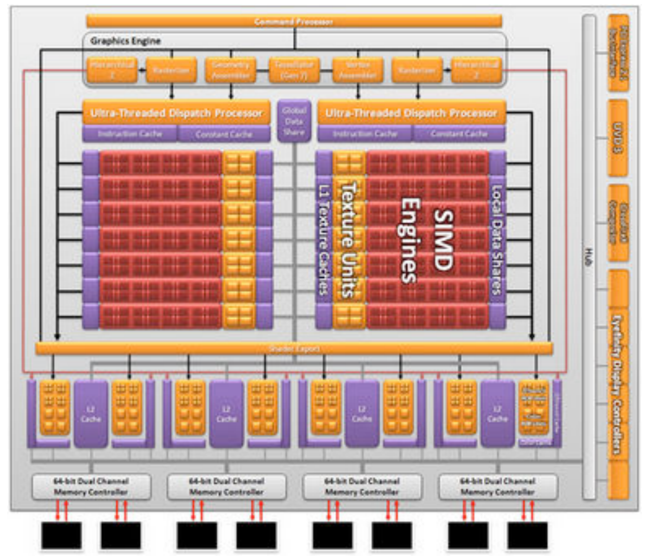
Language Approach

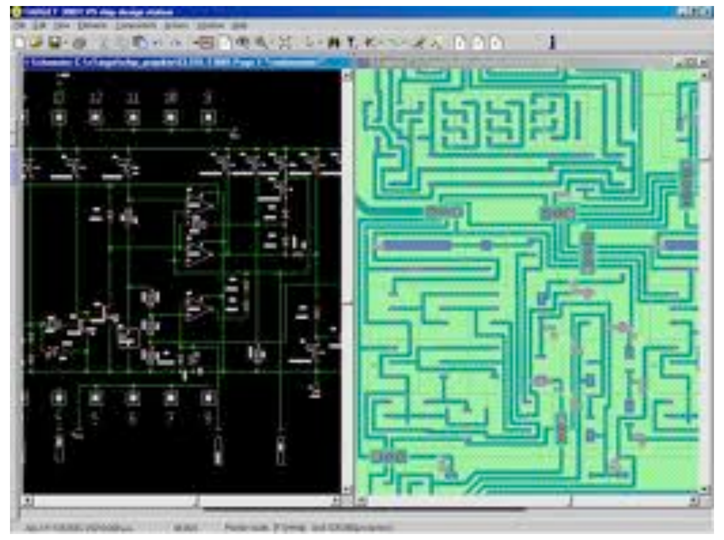
New Application/Legacy Code

Parallel Language









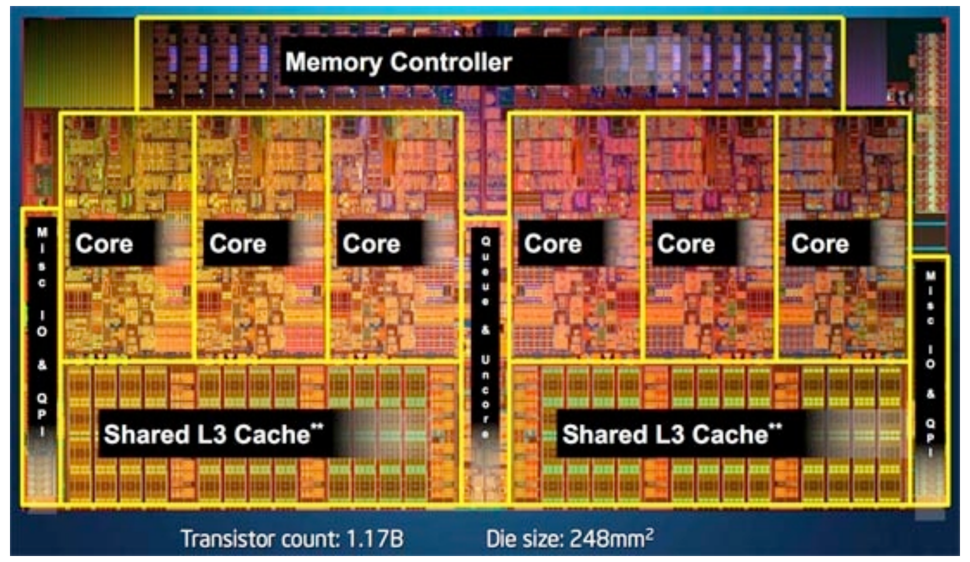
Language Approach

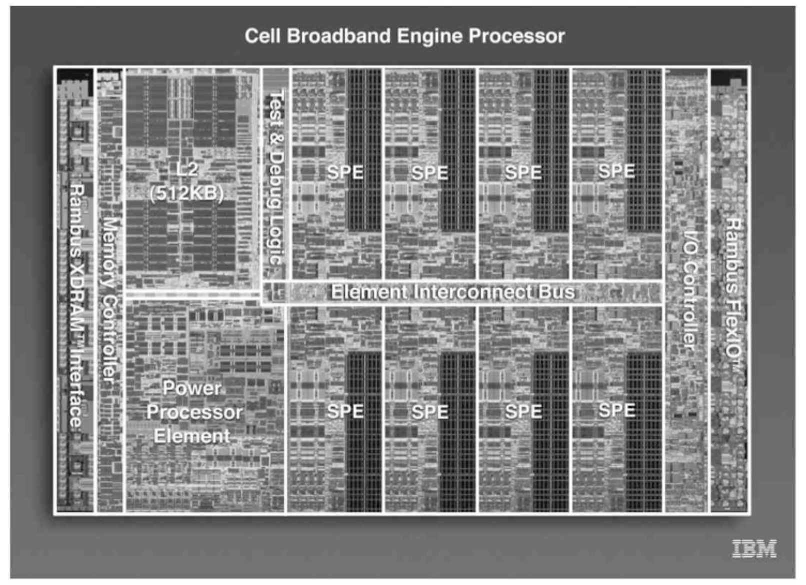
New Application/Legacy Code

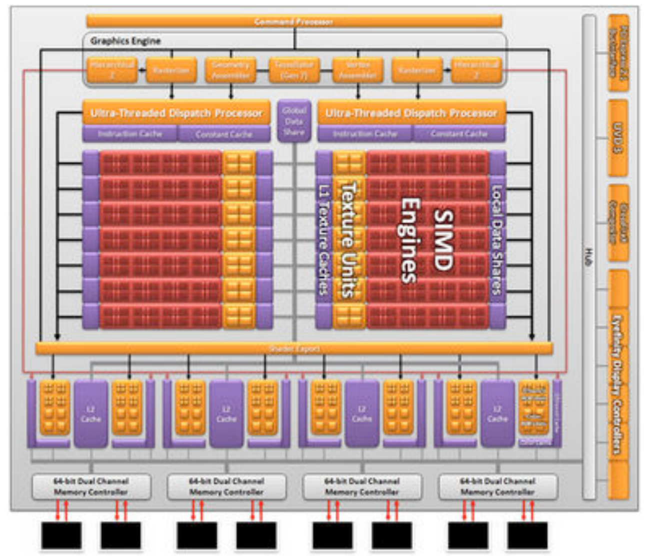
User rewrites

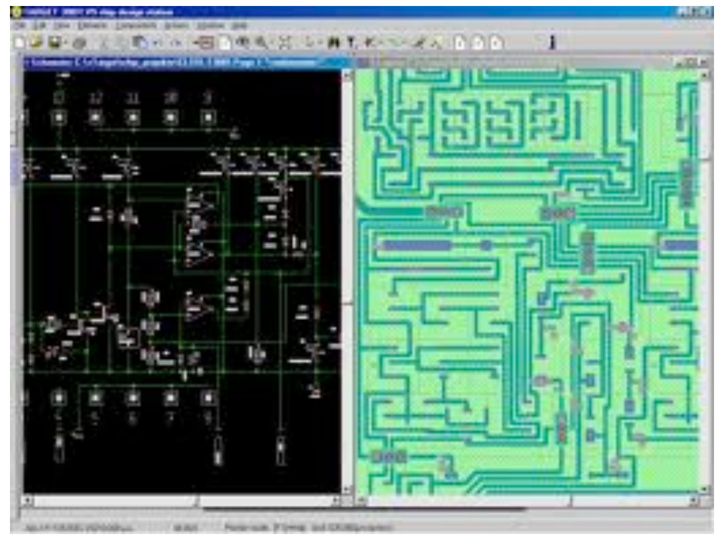
Parallel Language

Write new compiler









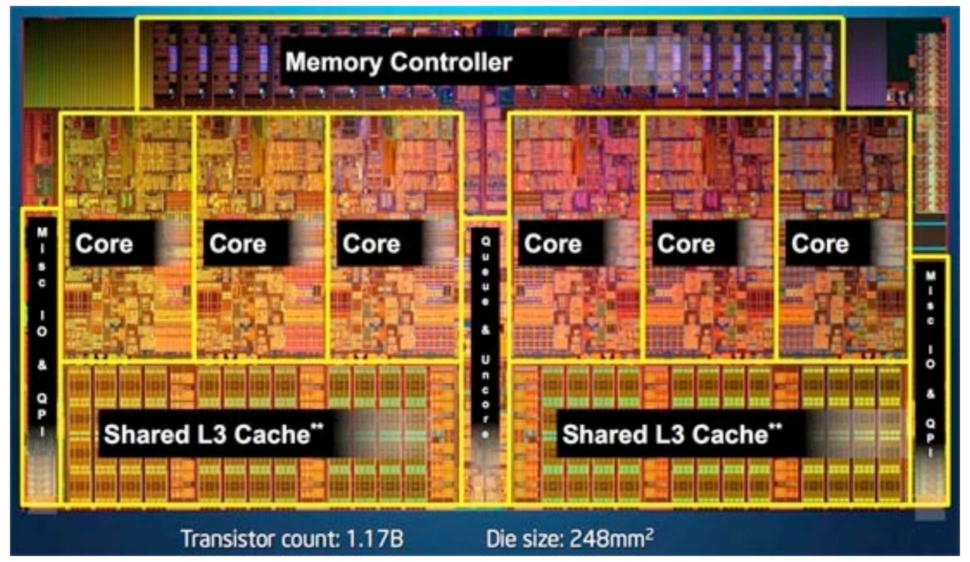
Language Approach

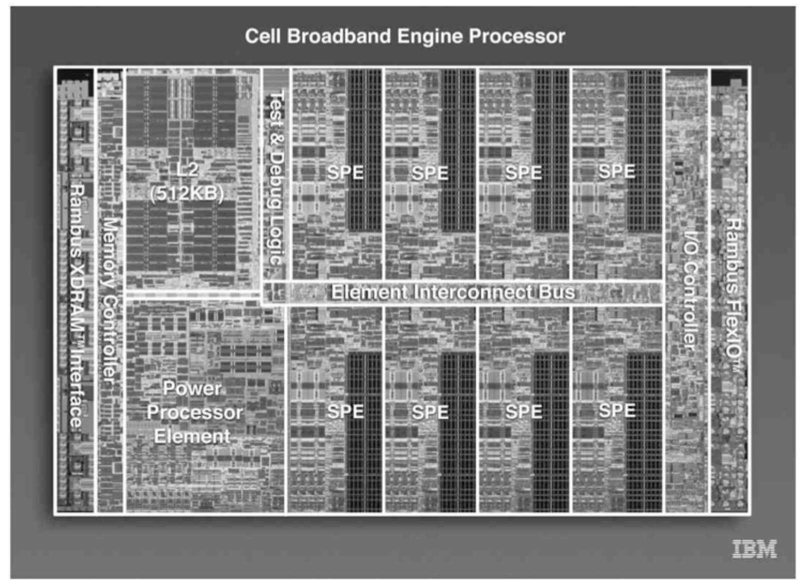
New Application/Legacy Code

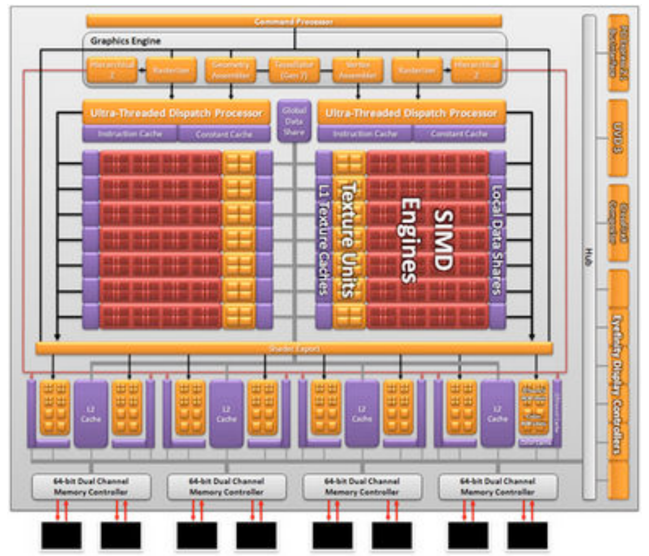
User rewrites

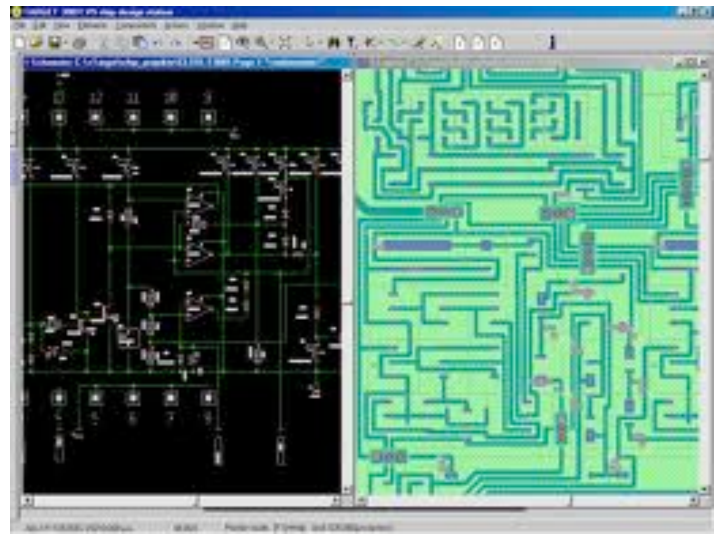
Parallel Language

Write new
compiler







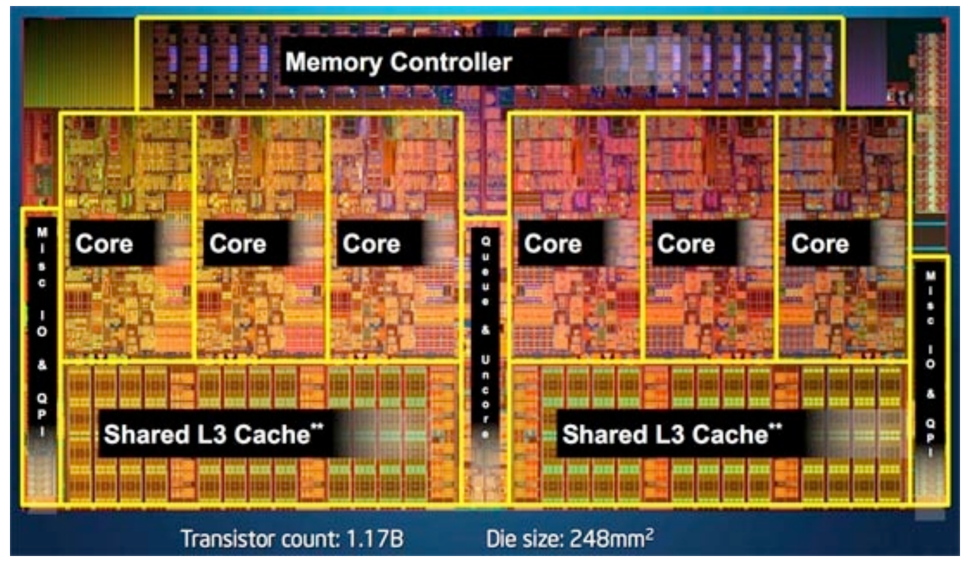


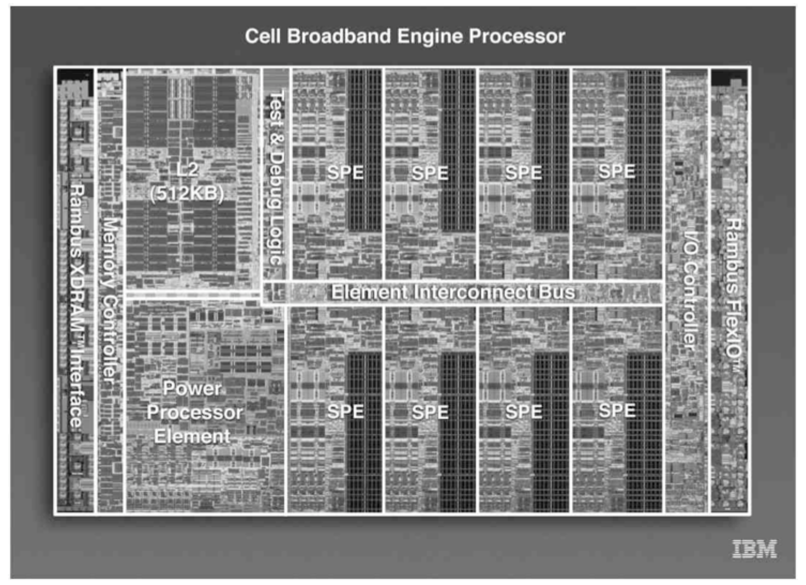
A universal parallel language
+ opt compiler per ISA/platform + smart runtime/glue?

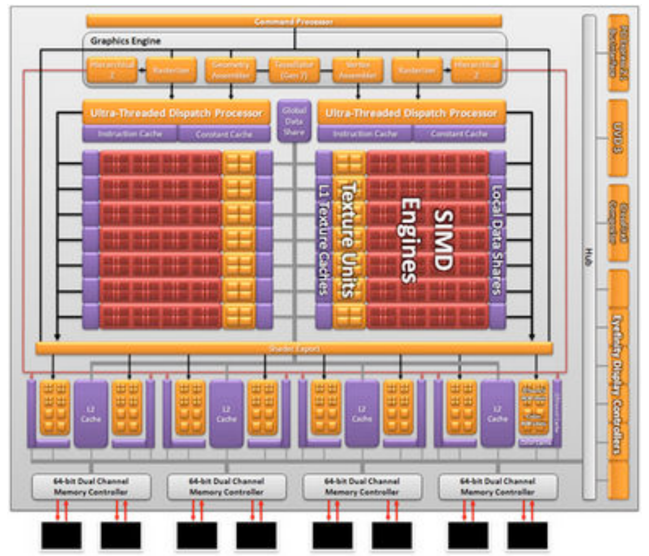
DSL approach

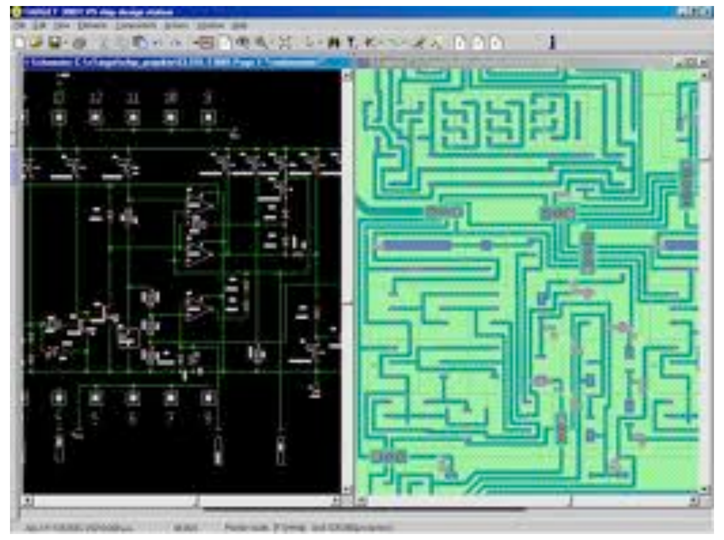
New Application/Legacy Code

DSL DSL DSL DSL



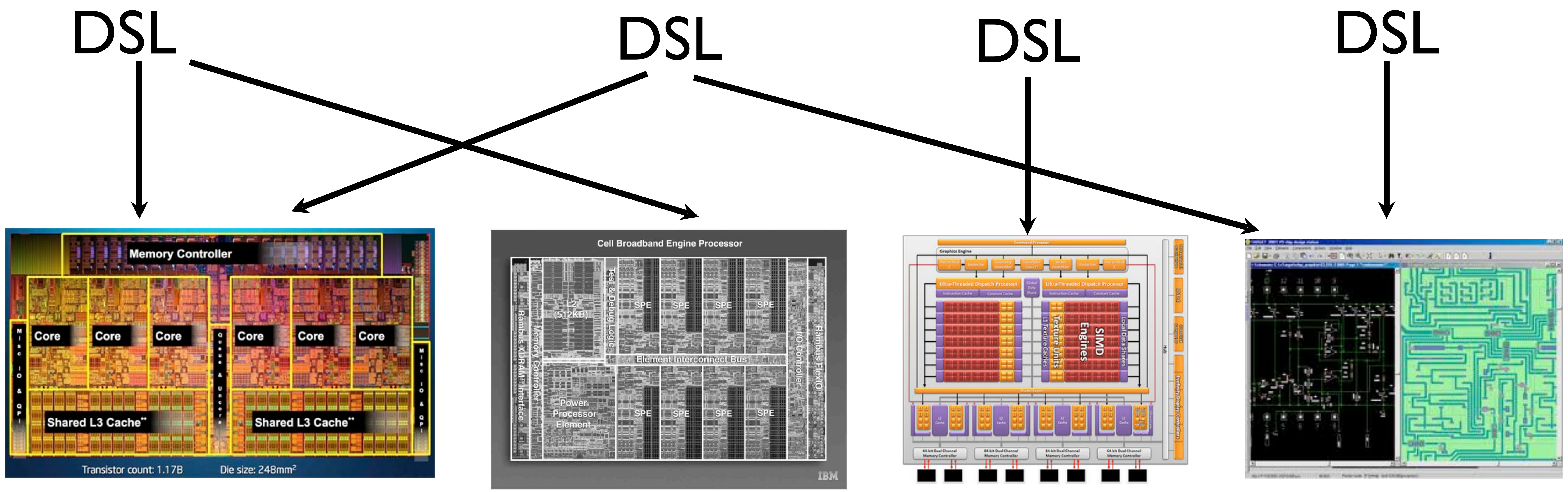




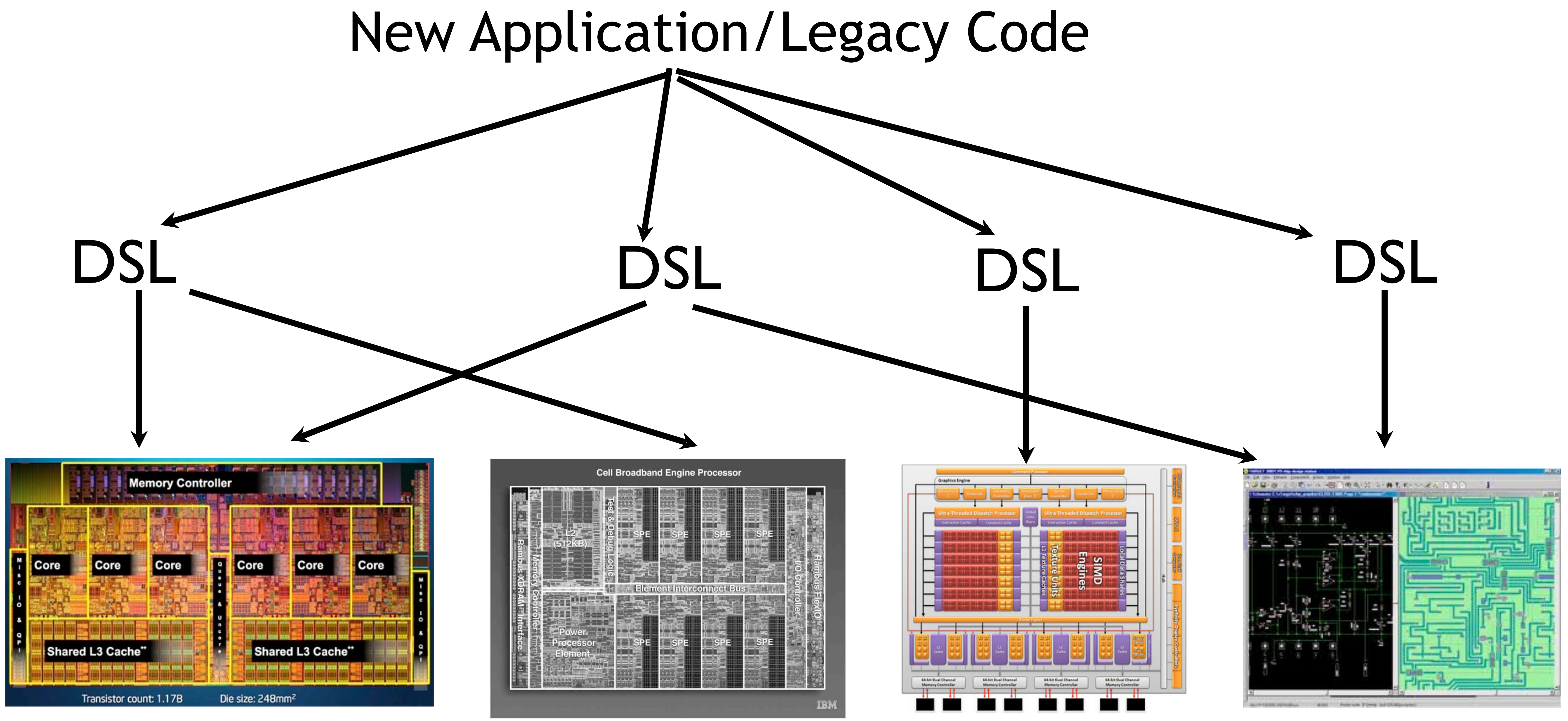


DSL approach

New Application/Legacy Code



DSL approach



Many specialised languages

+ rewrite and hope it works on your (next) machine?

Good performance is hard to get even with
well defined parallel language CUDA/OpenCL

GPU-Accelerated Libraries

GPU-Accelerated libraries provide highly-optimized algorithms and functions you can incorporate into your applications, with minimal changes to your existing code. Many support drop-in compatibility to replace industry standard CPU-only libraries such as MKL, IPP, FFTW and widely-used libraries. Some also feature automatic multi-GPU performance scaling.

AmgX
A simple path to accelerated core solvers, providing up to 10x acceleration in the computationally intense linear solver portion of simulations, and is very well suited for implicit unstructured methods.

cuDNN
NVIDIA cuDNN is a GPU-accelerated library of primitives for deep neural networks. It is designed to be integrated into higher-level machine learning frameworks.

cuFFT
NVIDIA CUDA Fast Fourier Transform Library (cuFFT) provides a simple interface for computing FFTs up to 10x faster, without having to develop your own custom GPU FFT implementation.

IndeX Framework
NVIDIA IndeX Framework is a real-time scalable visualization plug-in for ParaView.

cuRAND
The CUDA Random Number Generation library performs high quality GPU-accelerated random number generation (RNG) over 8x faster than typical CPU only code.

CUDA Math Library
An industry proven, highly accurate collection of standard mathematical functions, providing high performance on NVIDIA GPUs.

Thrust
A powerful, open source library of parallel algorithms and data structures. Perform GPU-accelerated sort, scan, transform, and reductions with just a few lines of code.

NVBIO
A GPU-accelerated C++ framework for High-Throughput Sequence Analysis for both short and long read alignment.

nvGRAPH
nvGRAPH Analytics Library is a GPU-accelerated graph analytics library.

GIE
NVIDIA GPU Inference Engine is a high performance neural network inference library for deep learning applications

NPP
NVIDIA Performance Primitives is a GPU accelerated library with a very large collection of 1000's of image processing primitives and signal processing primitives.

FFmpeg
FFmpeg is a popular open-sour multi-media framework with a library of plugins that can be applied to various parts of the audio and video processing pipelines.

NVIDIA VIDEO CODEC SDK
Accelerate video compression with the NVIDIA Video Codec SDK. This SDK includes documentation and code samples that illustrate how to use NVIDIA's NVENC and NVDEC hardware in GPUs to accelerate encode, decode, and transcode of H.264 and HEVC video formats.

HiPLAR
HiPLAR (High Performance Linear Algebra in R) delivers high performance linear algebra (LA) routines for the R platform for statistical computing using the latest software libraries for heterogeneous architectures.

OpenCV
OpenCV is the leading open source library for computer vision, image processing and machine learning, and now features GPU acceleration for real-time operation.

Geometry Performance Primitives(GPP)
GPP is a computational geometry engine that is optimized for GPU acceleration, and can be used in advanced Graphical Information Systems (GIS), Electronic Design Automation (EDA), computer vision, and motion planning solutions.

CHOLMOD
GPU-accelerated CHOLMOD is part of the SuiteSparse linear algebra package by Prof. Tim Davis. SuiteSparse is used extensively throughout industry and academia.

CULA Tools
GPU-accelerated linear algebra library by EM Photonics, that utilizes CUDA to dramatically improve the computation speed of sophisticated mathematics.

MAGMA
A collection of next gen linear algebra routines. Designed for heterogeneous GPU-based architectures. Supports current LAPACK and BLAS standards.

IMSL Fortran Numerical Library
Developed by RogueWave, a comprehensive set of mathematical and statistical functions that offloads work to GPUs.

aralution
library for sparse iterative ethods with special focus on ulti-core and accelerator chnology such as GPUs.

Triton Ocean SDK
Triton provides real-time visual simulation of the ocean and bodies of water for games, simulation, and training applications.

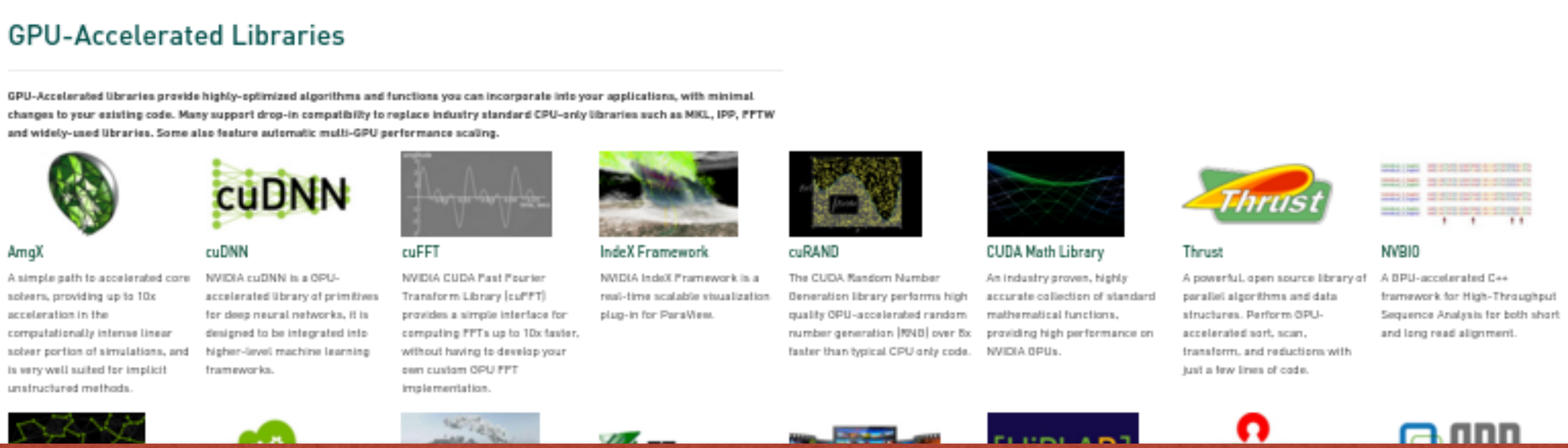
cuBLAS
NVIDIA CUDA BLAS Library (cuBLAS) is a GPU-accelerated version of the complete standard BLAS library that delivers 6x to 17x faster performance than the latest MKL BLAS.

ArrayFire
Comprehensive, open source GPU function library. Includes functions for math, signal and image processing, statistics, and many more. Interfaces for C, C++, Java, R and Fortran.

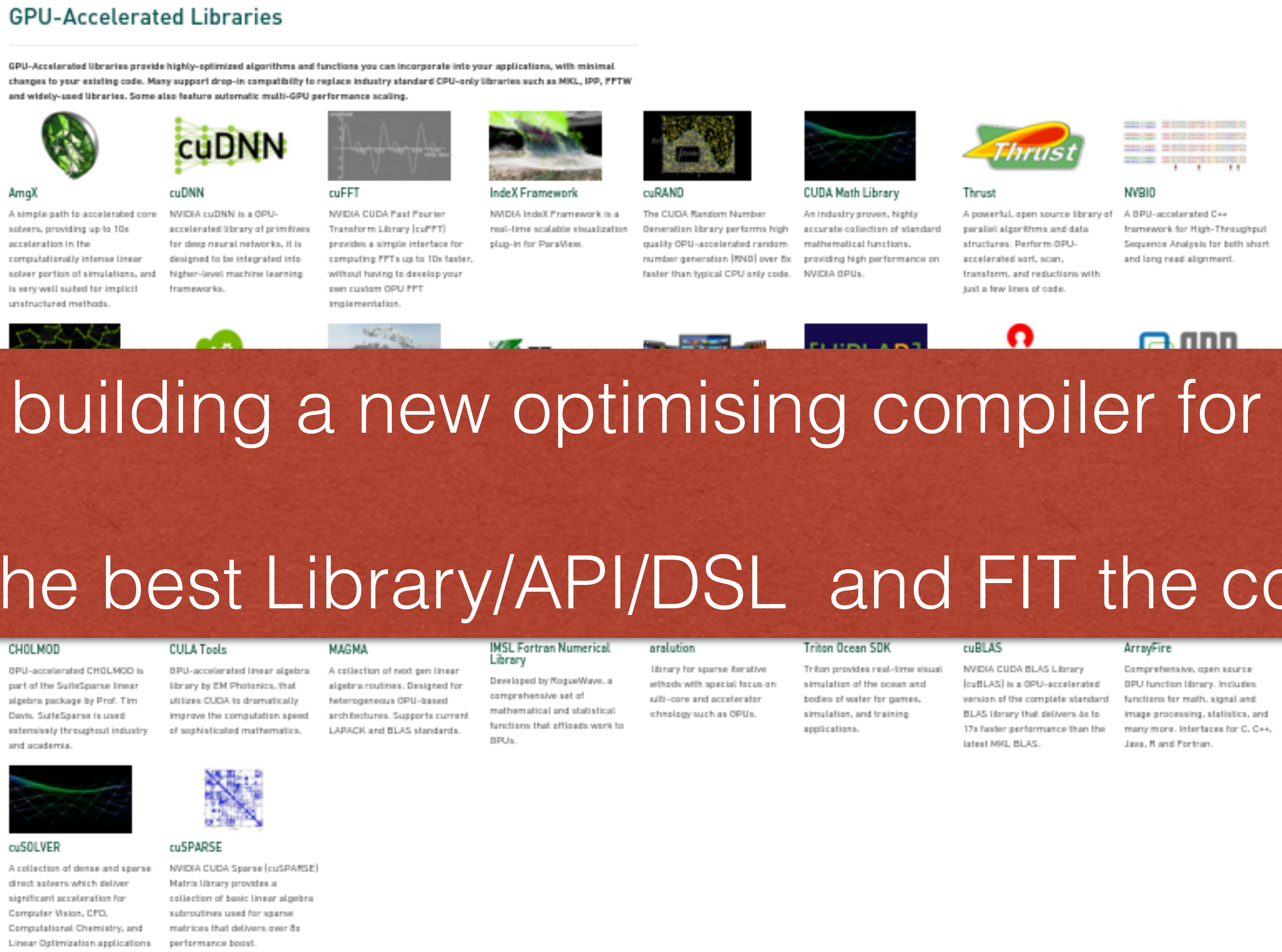
cuSOLVER
A collection of dense and sparse direct solvers which deliver significant acceleration for Computer Vision, CFD, Computational Chemistry, and Linear Optimization applications

cuSPARSE
NVIDIA CUDA Sparse (cuSPARSE) Matrix library provides a collection of basic linear algebra subroutines used for sparse matrices that delivers over 8x performance boost.

Good performance is hard to get even with well defined parallel language CUDA/OpenCL



Rather than building a new optimising compiler for each platform



GPU-Accelerated Libraries

GPU-Accelerated libraries provide highly-optimized algorithms and functions you can incorporate into your applications, with minimal changes to your existing code. Many support drop-in compatibility to replace industry standard CPU-only libraries such as MKL, IPP, FFTW and widely-used libraries. Some also feature automatic multi-GPU performance scaling.

AmgX
A simple path to accelerated core solvers, providing up to 10x acceleration in the computationally intense linear solver portion of simulations, and is very well suited for implicit unstructured methods.

cuDNN
NVIDIA cuDNN is a GPU-accelerated library of primitives for deep neural networks. It is designed to be integrated into higher-level machine learning frameworks.

cuFFT
NVIDIA CUDA Fast Fourier Transform Library (cuFFT) provides a simple interface for computing FFTs up to 10x faster, without having to develop your own custom GPU FFT implementation.

IndeX Framework
NVIDIA IndeX Framework is a real-time scalable visualization plug-in for ParaView.

cuRAND
The CUDA Random Number Generation library performs high quality GPU-accelerated random number generation (RNG) over 8x faster than typical CPU only code.

CUDA Math Library
An industry proven, highly accurate collection of standard mathematical functions, providing high performance on NVIDIA GPUs.

Thrust
A powerful, open source library of parallel algorithms and data structures. Perform GPU-accelerated sort, scan, transform, and reductions with just a few lines of code.

NVBIO
A GPU-accelerated C++ framework for High-Throughput Sequence Analysis for both short and long read alignment.

Rather than building a new optimising compiler for each platform

Pick the best Library/API/DSL and FIT the code to it

CHOLMOD
GPU-accelerated CHOLMOD is part of the SuiteSparse linear algebra package by Prof. Tim Davis. SuiteSparse is used extensively throughout industry and academia.

CULA Tools
GPU-accelerated linear algebra library by EM Photonics, that utilizes CUDA to dramatically improve the computation speed of sophisticated mathematics.

MAGMA
A collection of next gen linear algebra routines. Designed for heterogeneous GPU-based architectures. Supports current LAPACK and BLAS standards.

IMSL Fortran Numerical Library
Developed by RogueWave, a comprehensive set of mathematical and statistical functions that offloads work to GPUs.

aralution
library for sparse iterative ethods with special focus on ulti-core and accelerator chnology such as GPUs.

Triton Ocean SDK
Triton provides real-time visual simulation of the ocean and bodies of water for games, simulation, and training applications.

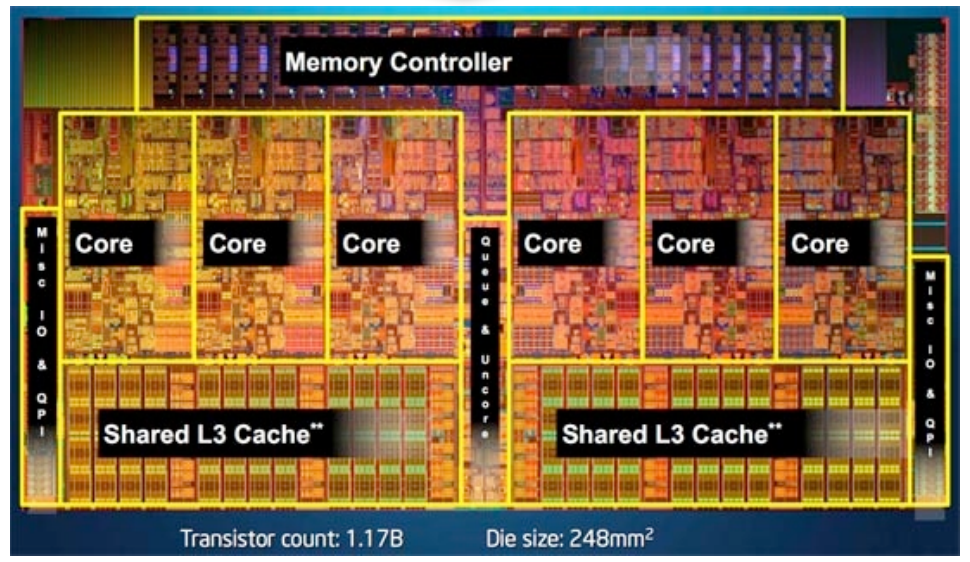
cuBLAS
NVIDIA CUDA BLAS Library (cuBLAS) is a GPU-accelerated version of the complete standard BLAS library that delivers 6x to 17x faster performance than the latest MKL BLAS.

ArrayFire
Comprehensive, open source GPU function library. Includes functions for math, signal and image processing, statistics, and many more. Interfaces for C, C++, Java, R and Fortran.

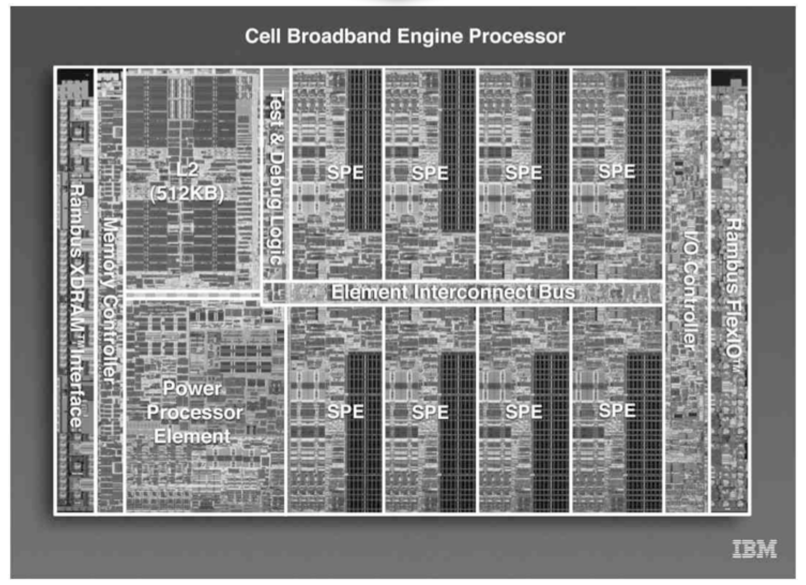
cuSOLVER
A collection of dense and sparse direct solvers which deliver significant acceleration for Computer Vision, CFD, Computational Chemistry, and Linear Optimization applications

cuSPARSE
NVIDIA CUDA Sparse (cuSPARSE) Matrix library provides a collection of basic linear algebra subroutines used for sparse matrices that delivers over 8x performance boost.

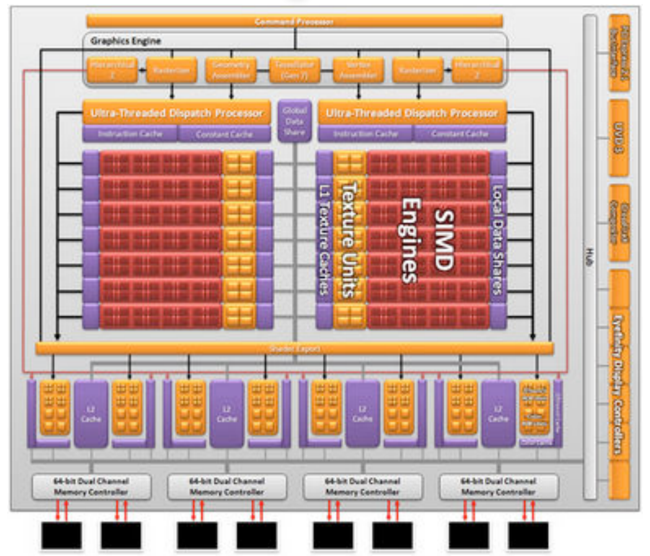
Legacy Program



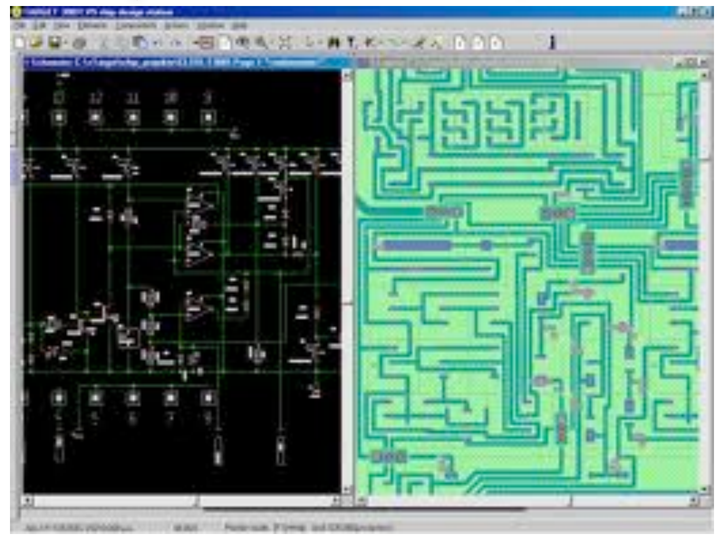
pthreads



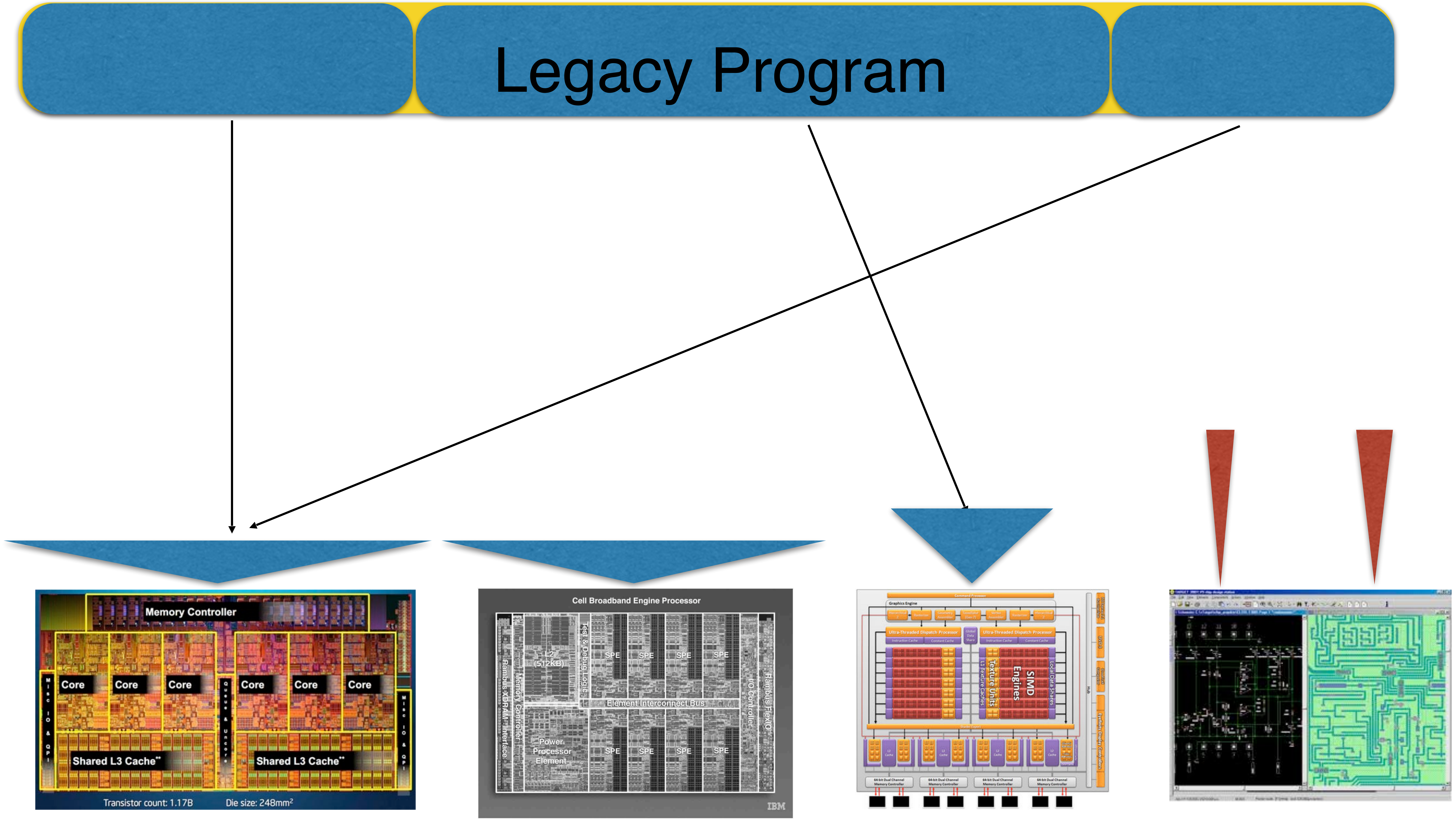
multi C



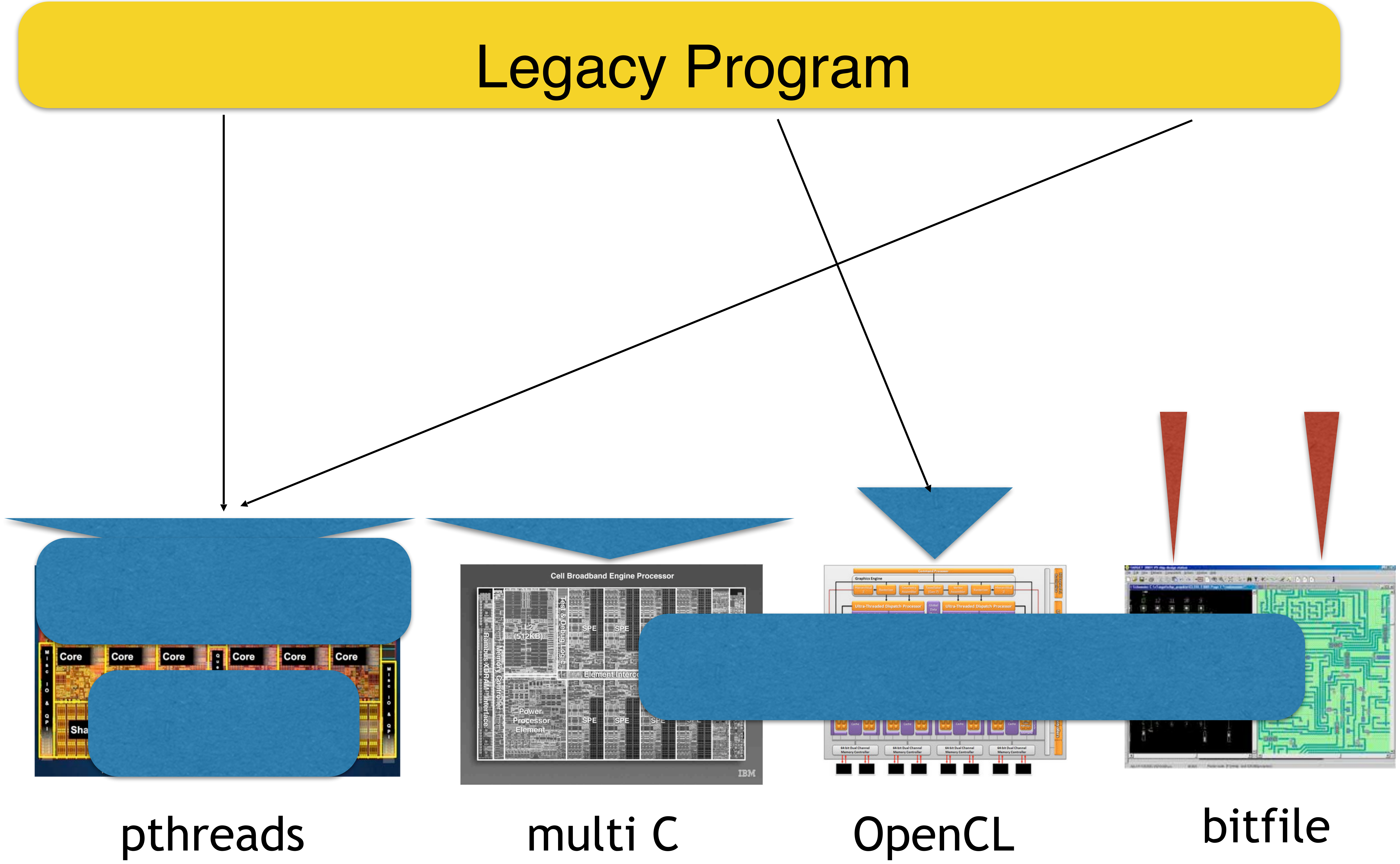
OpenCL



bitfile

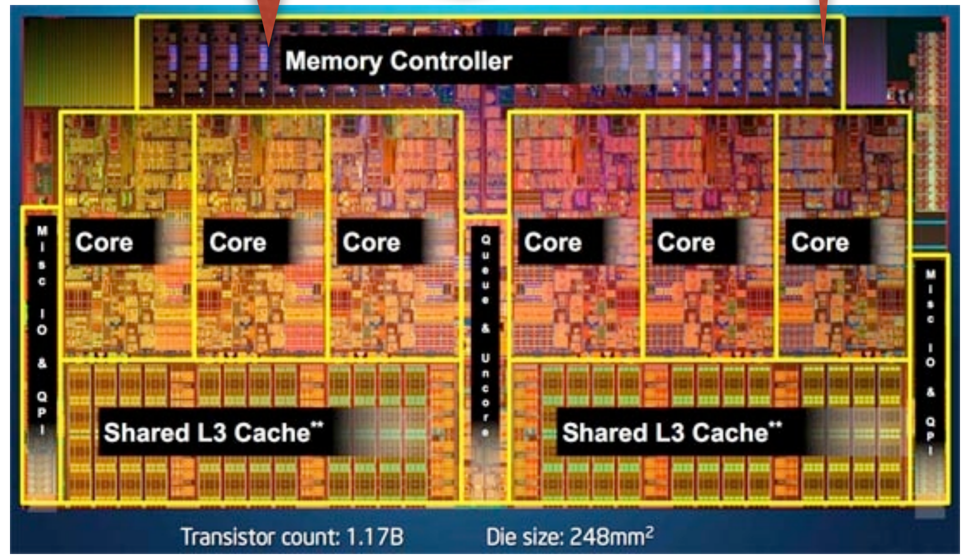


pthreads multi C OpenCL bitfile

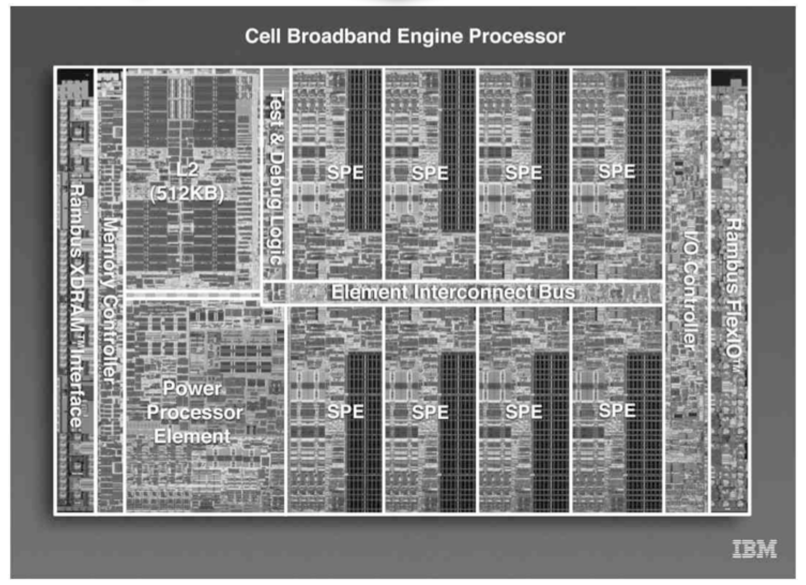

Legacy Program
Core
Core
Core
Core
Core
Core
Cell Broadband Engine Processor
IBM
pthreads
multi C
OpenCL
bitfile

Legacy Program

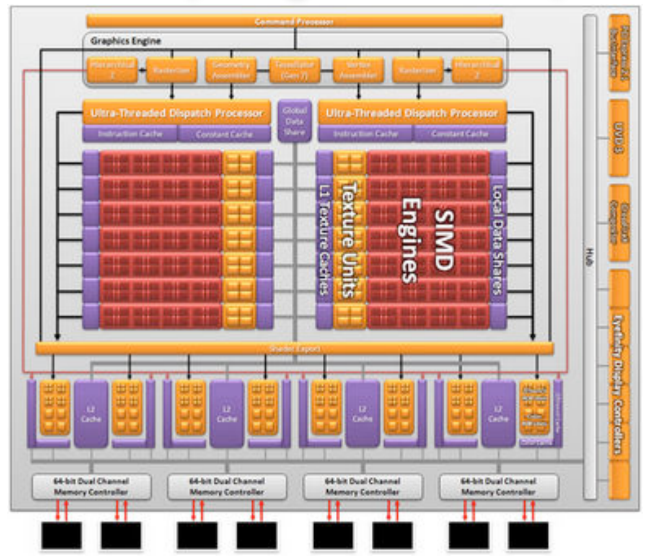
DSL/ Library/ API



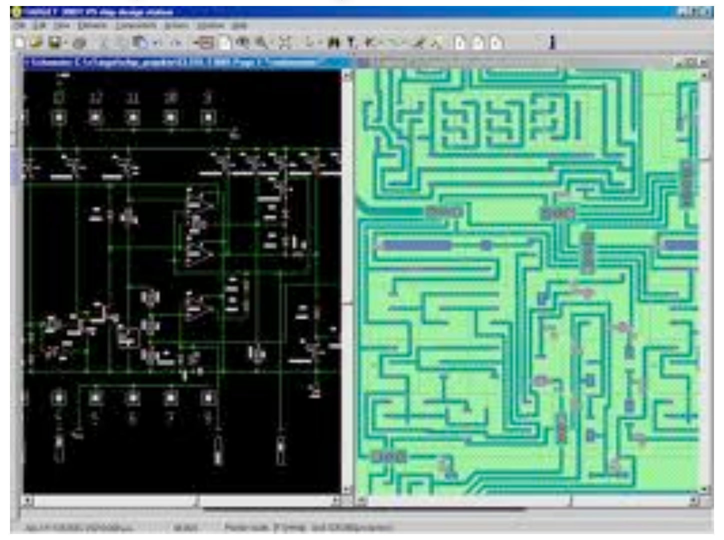
Polly TBB
BLAS



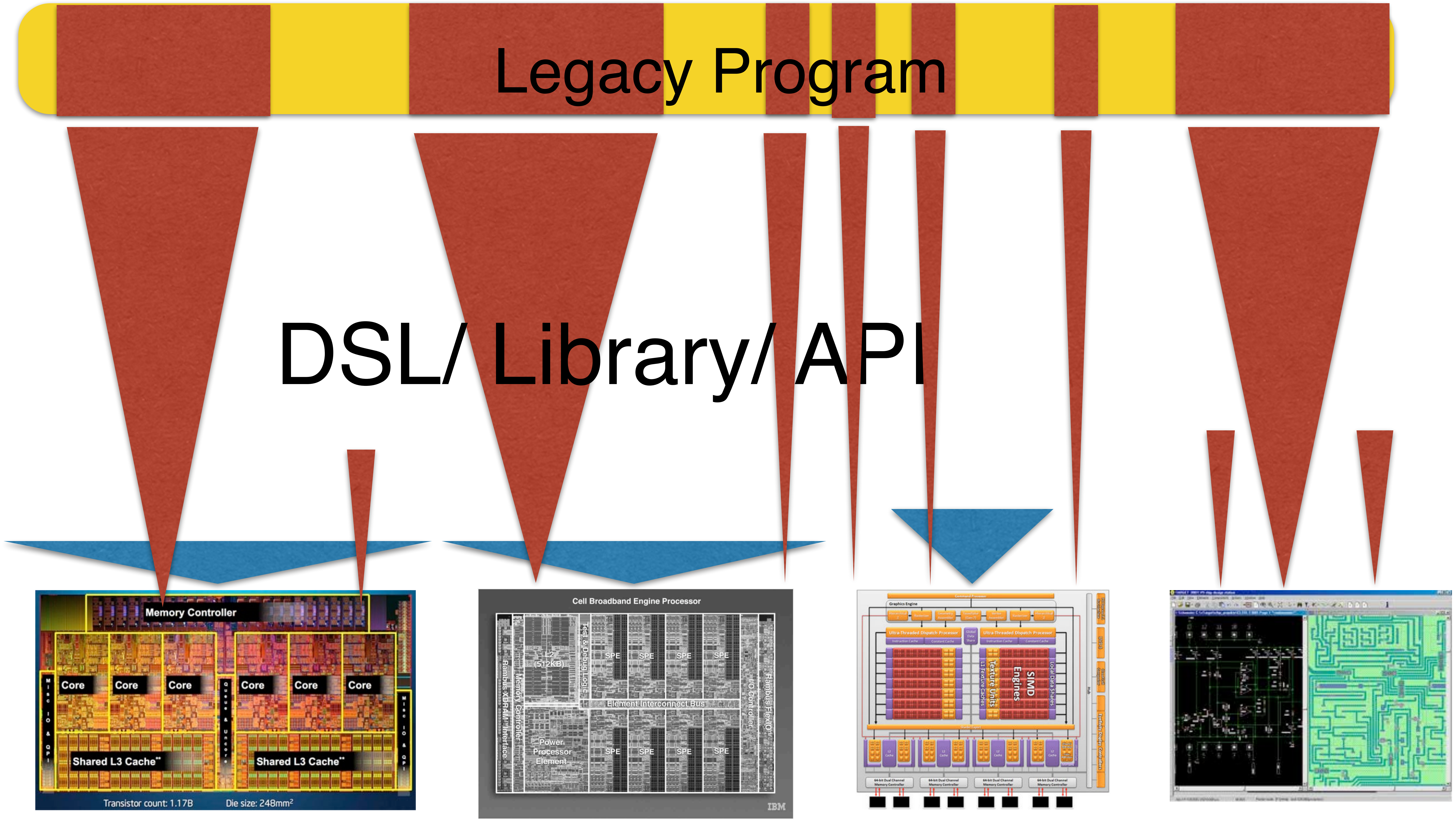
Milk
Halide



PolyACC Lift
OpenGL



fir fft



Polly TBB
BLAS

Milk
Halide

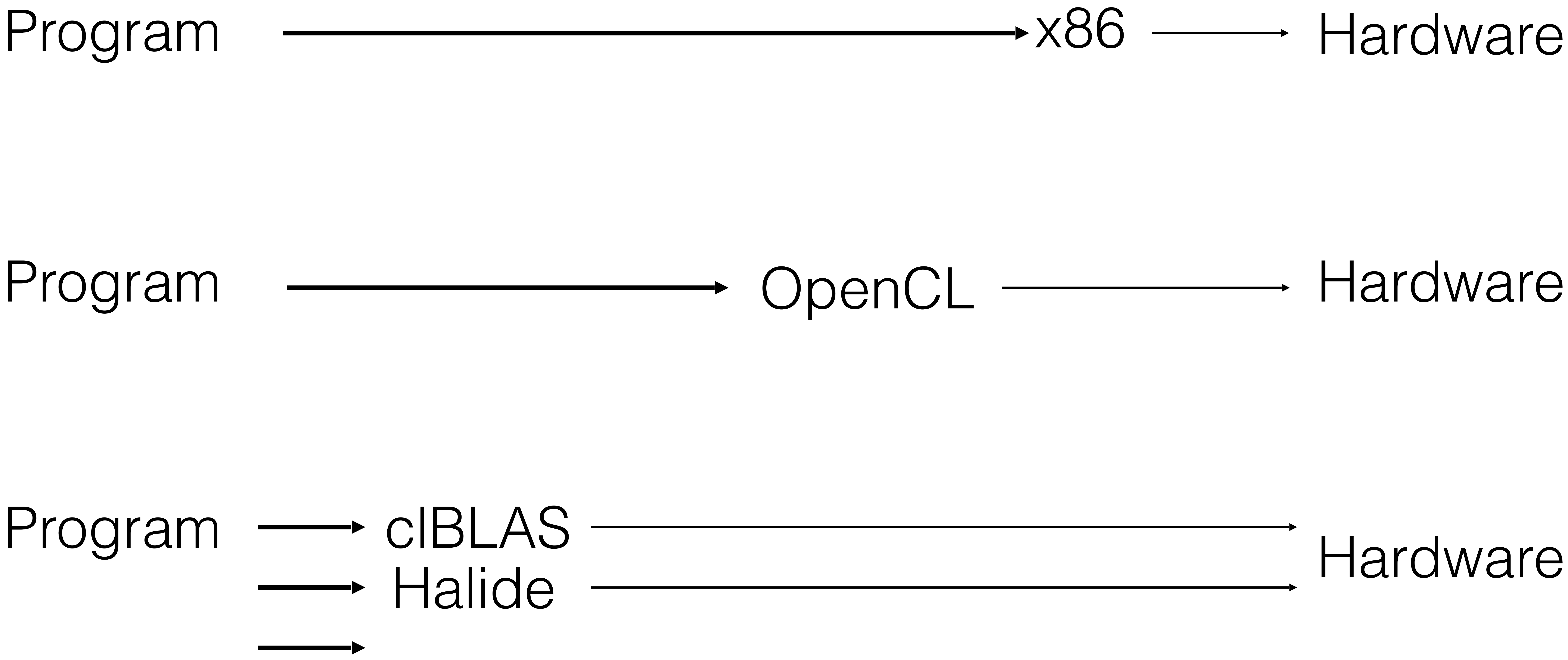
PolyACC Lift
OpenGL

fir fft

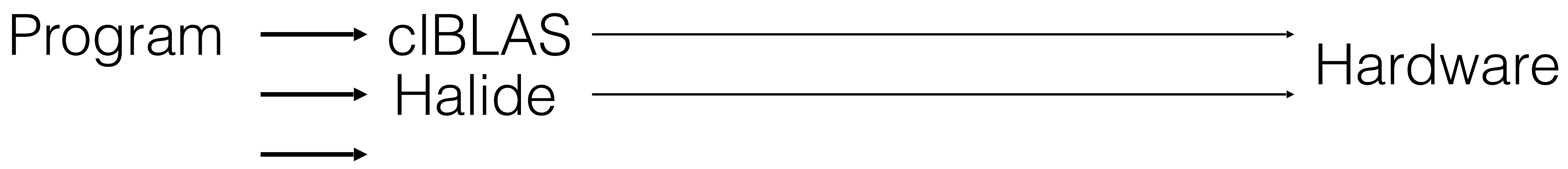
Program ⟶ x86 → Hardware

Program → x86 → Hardware

Program → OpenCL → Hardware


Program
x86
Hardware
Program
OpenCL
Hardware
Program
clBLAS
Halide
Hardware

+ Target nearer to algorithm
+ Target willl always perform well



+ Target nearer to algorithm
+ Target willl always perform well
- Target complex and changeable

Program → clBLAS → Hardware

Program → Halide → Hardware

Program →

+ Target nearer to algorithm
+ Target willl always perform well
- Target complex and changeable

Program → clBLAS → Hardware
Program → Halide → Hardware
Program →

Constant change means any solution must work for any API, any DSL

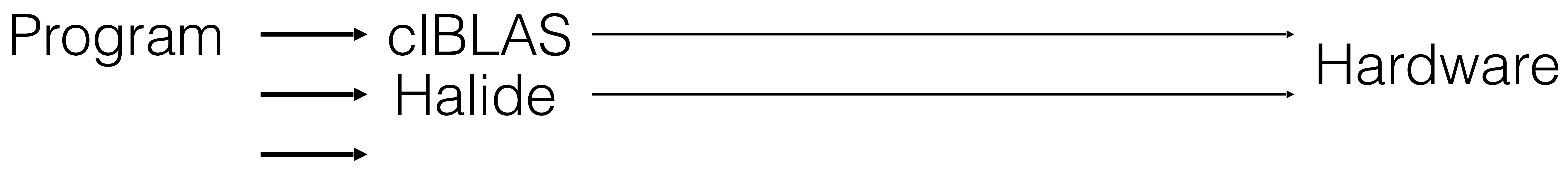
Need to automate

\+ Target nearer to algorithm

\+ Target willl always perform well

\- Target complex and changeable

\- Target may be at higher level

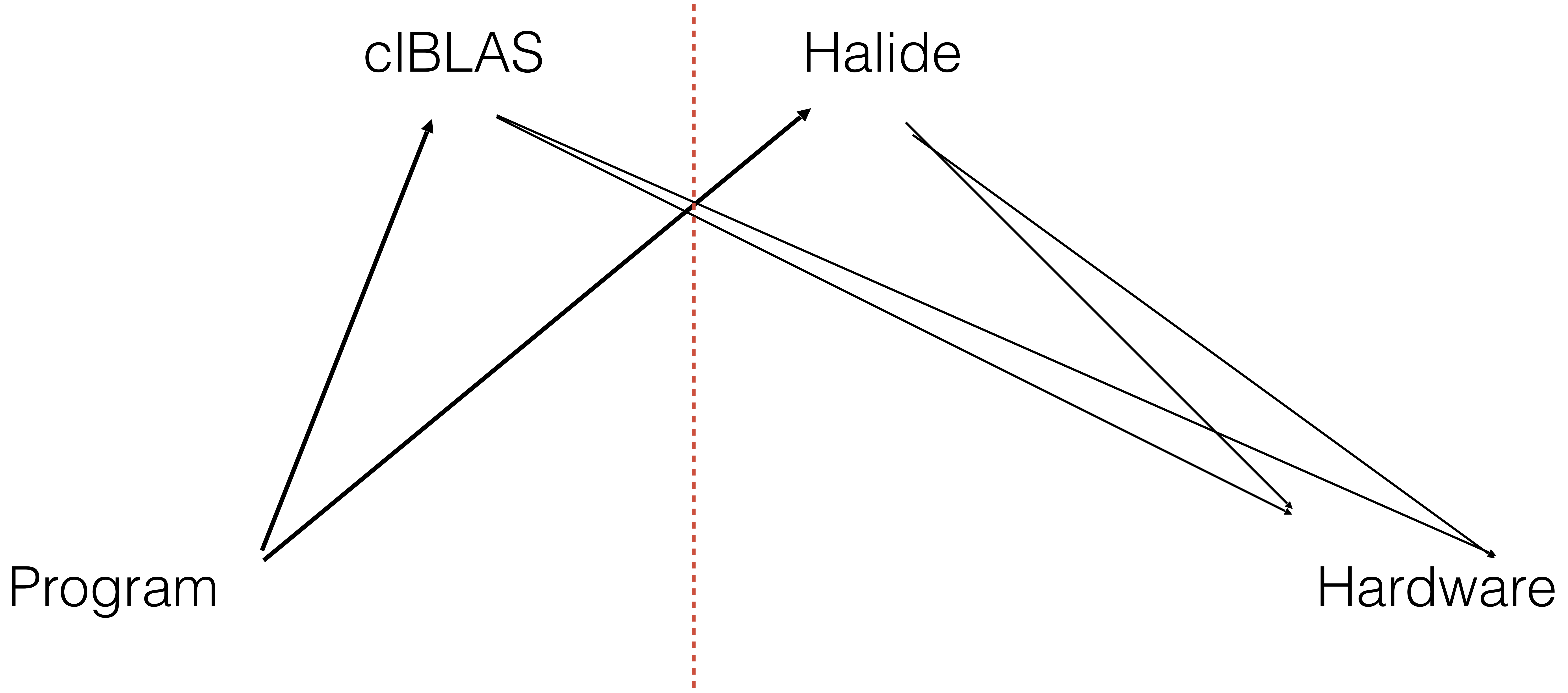


Rather than compile code to hardware

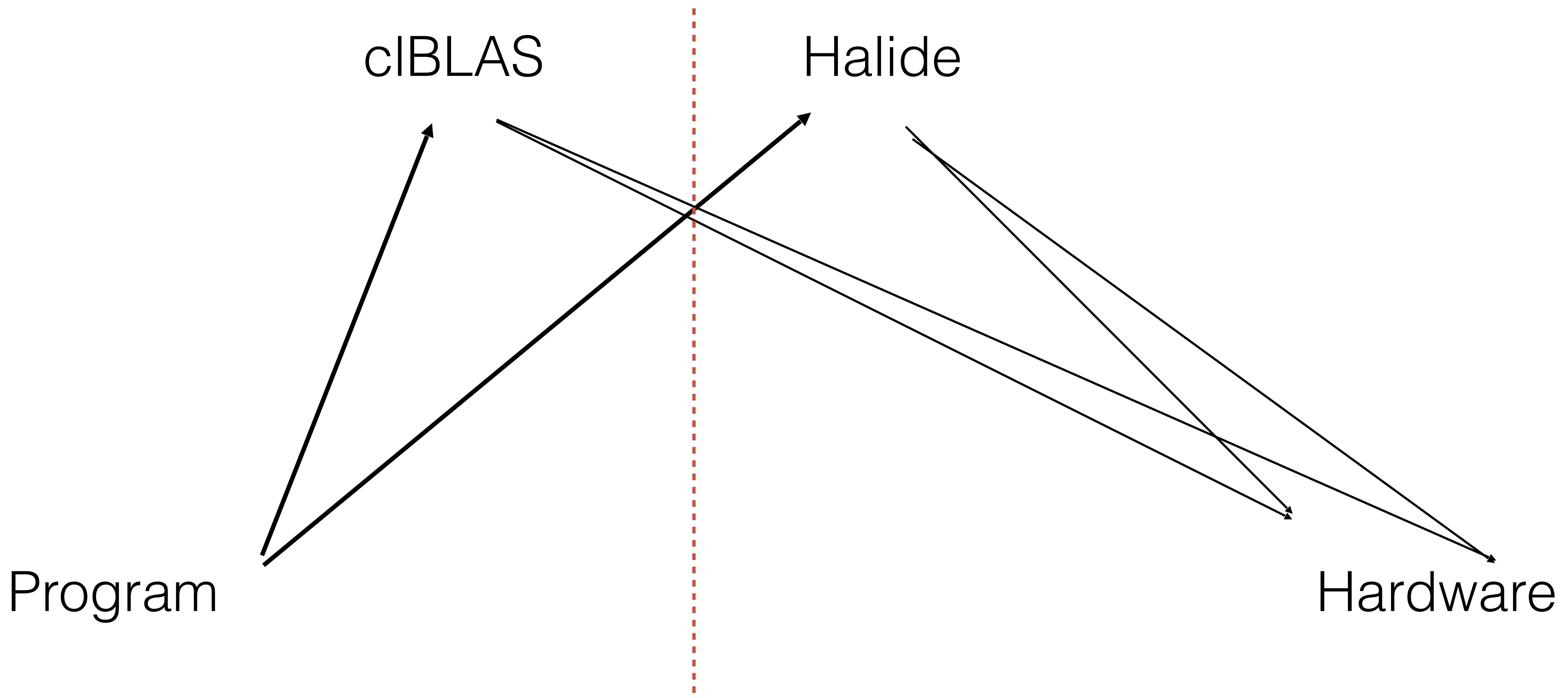
By lowering code for each language and each ISA

Program ⟶ Hardware

Instead LIFT code to API or DSL



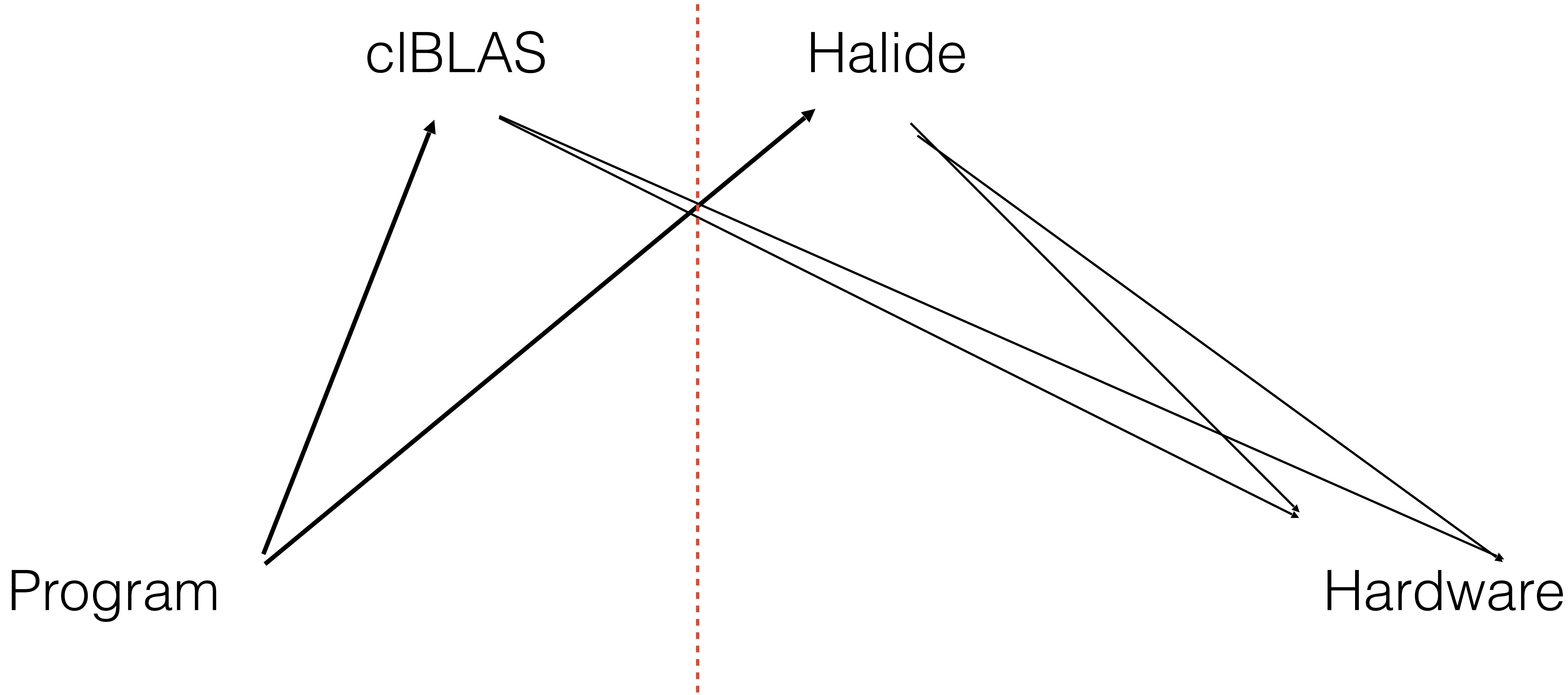
LIFT code to API or DSL



Vendor responsibility to map API/DSL to hardware - already the case

Our job - automatically lift it to API/DSL enabling hardware utilisation

LIFT code to API or DSL



How is API/DSL described?
How is matching code discovered?
How is code replaced/translated?

Well known things

My view

Concrete results

Can we go further ?

Summary

5 approaches to lifting

Search using constraints over LLVM IR: IDL+CanDL [18-20]
- targetted APIs in C/Fortran - dense/sparse linear algebra

Black-box Program Synthesis [19-21]
- eliminated need for writing constraints

API matching via IO behavioural equivalence [21-23]
- more robust detection

Neural Compilation [21-?]
- language to assembler translation using NMT/transformer

Program Lifting [22-?]
- beyond APIs lifting to DSLs/MLIR

5 approaches to lifting

Search using constraints over LLVM IR: IDL+CanDL [18-20]
- targetted APIs in C/Fortran - dense/sparse linear algebra

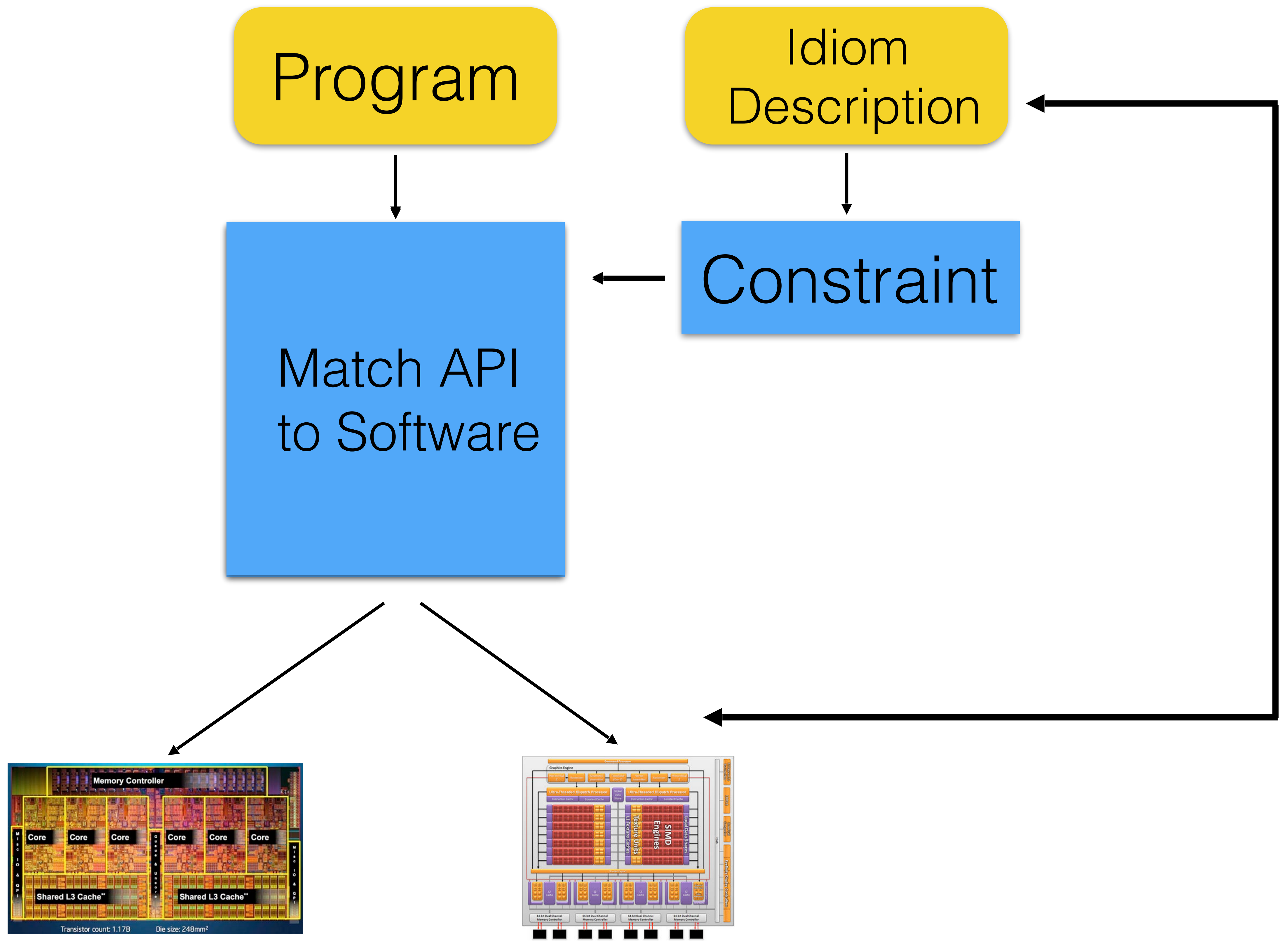
Black-box Program Synthesis [19-21]
- eliminated need for writing constraints

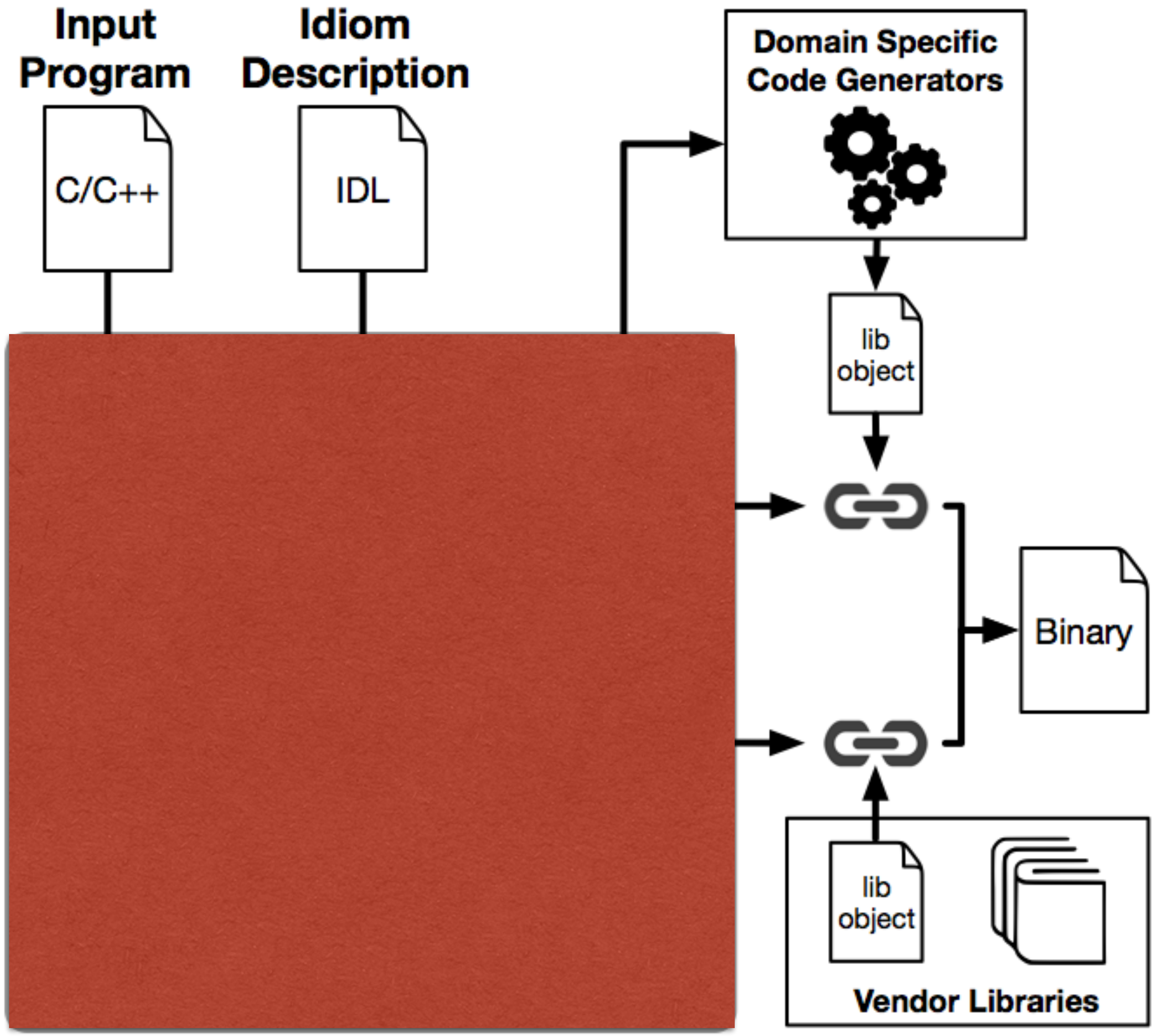
API matching via IO behavioural equivalence [21-23]
- more robust detection

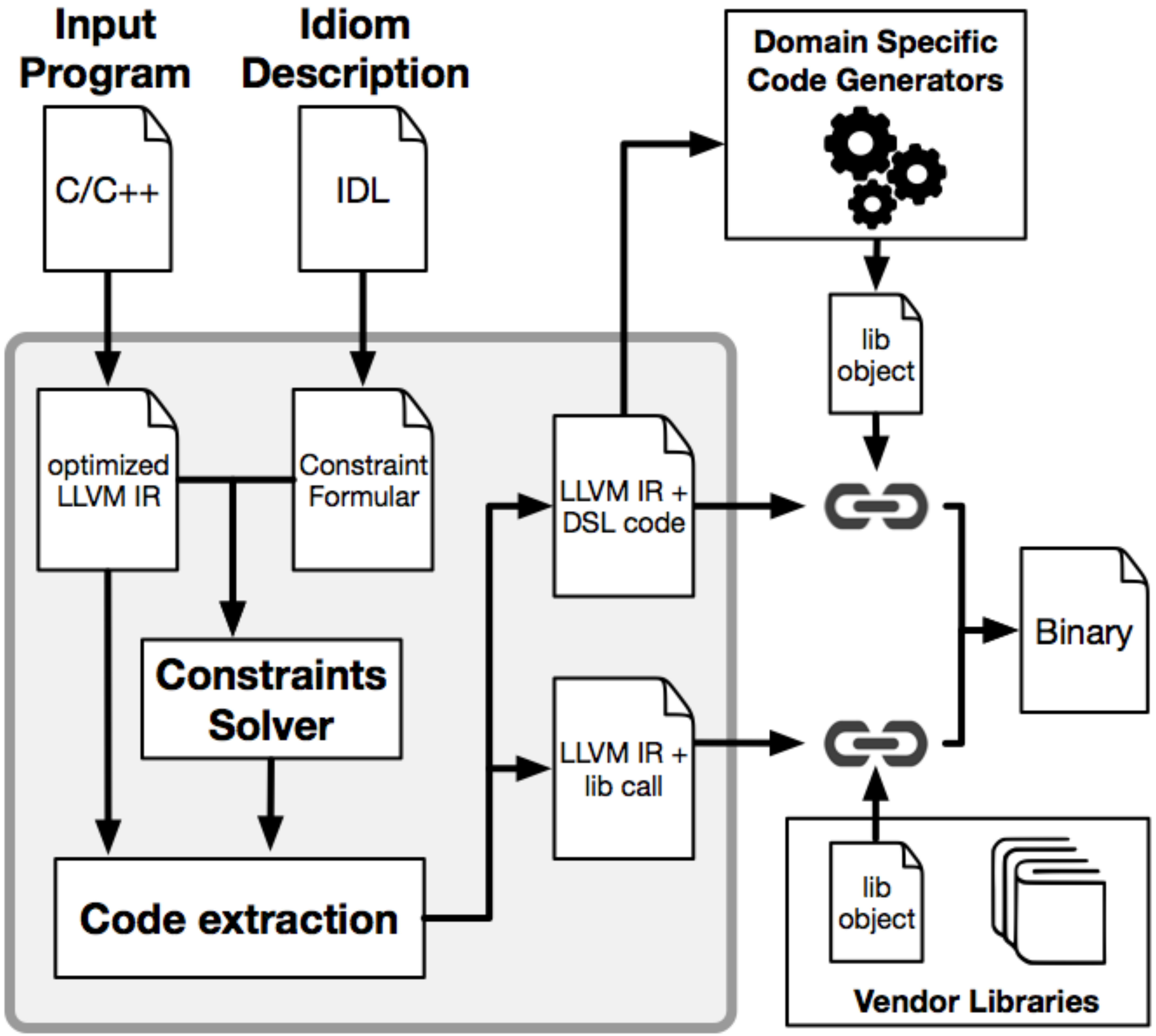
Neural Compilation [21-?]
- language to assembler translation using NMT/transformer

Program Lifting [22-?]
- beyond APIs lifting to DSLs/MLIR

Detect code structures that match interface




Input
Program
C/C++
Idiom
Description
IDL
Domain Specific
Code Generators
lib
object
Binary
lib
object
Vendor Libraries


Input Program
C/C++
Idiom Description
IDL
Domain Specific Code Generators
optimized LLVM IR
Constraint Formular
LLVM IR + DSL code
lib object
Constraints Solver
Binary
LLVM IR + lib call
Code extraction
lib object
Vendor Libraries

```c
for (j = 0; j < lastrow - firstrow + 1; j++) {
  sum = 0.0;
  for (k = rowstr[j]; k < rowstr[j+1]; k++) {
    sum = sum + a[k]*p[colidx[k]];
  }
  q[j] = sum;
}
```

```
Constraint SPMV
( inherits For and
  inherits VectorStore
      with {iterator} as {idx}
       and {begin} as {begin} at {output} and
  inherits ReadRange
      with {iterator} as {idx}
       and {inner.iter_begin} as {range_begin}
       and {inner.iter_end}   as {range_end} and
  inherits For at {inner} and
  inherits VectorRead
      with {inner.iterator} as {idx}
       and {begin} as {begin} at {idx_read} and
  inherits VectorRead
      with {idx_read.value} as {idx}
       and {begin} as {begin} at {indir_read} and
  inherits VectorRead
      with {inner.iterator} as {idx}
       and {begin} as {begin} at {seq_read} and
  inherits DotProductLoop
      with {inner}             as {loop}
       and {indir_read.value} as {src1}
       and {seq_read.value}   as {src2}
       and {output.address}   as {update_address})
End
```

```c
#include "mkl.h"

// …
void spmv_csr_harness(int rows, int* ranges,
    int* indir, double* vector, double* matrix,
    double* output) {

  sparse_matrix_t A;
  // …

  struct matrix_descr C;
  C.type = SPARSE_MATRIX_TYPE_GENERAL;
  C.mode = SPARSE_FILL_MODE_LOWER;
  C.diag = SPARSE_DIAG_NON_UNIT;
  mkl_sparse_d_mv(SPARSE_OPERATION_NON_TRANSPOSE,
      1.0, A, D, vector, 0.0, output);
}
```

[CC20]

Speedup

Speedup over sequential code 1.1x to 250x

Automatically finds and exploits parallel idioms

[ASPLOS18]

5 approaches to lifting

Search using constraints over LLVM IR: IDL+CanDL [18-20]
- targetted APIs in C/Fortran - dense/sparse linear algebra

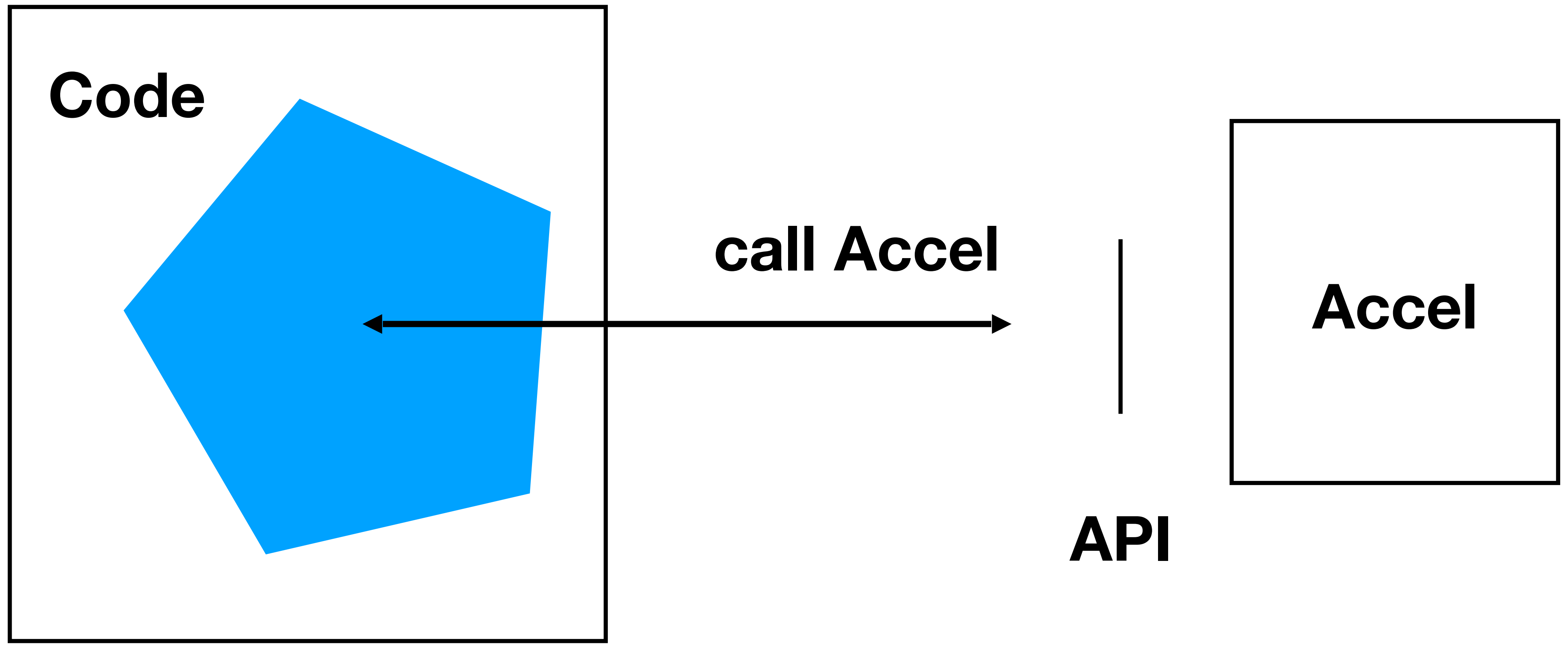
Black-box Program Synthesis [19-21]
- eliminated need for writing constraints

API matching via IO behavioural equivalence [21-23]
- more robust detection

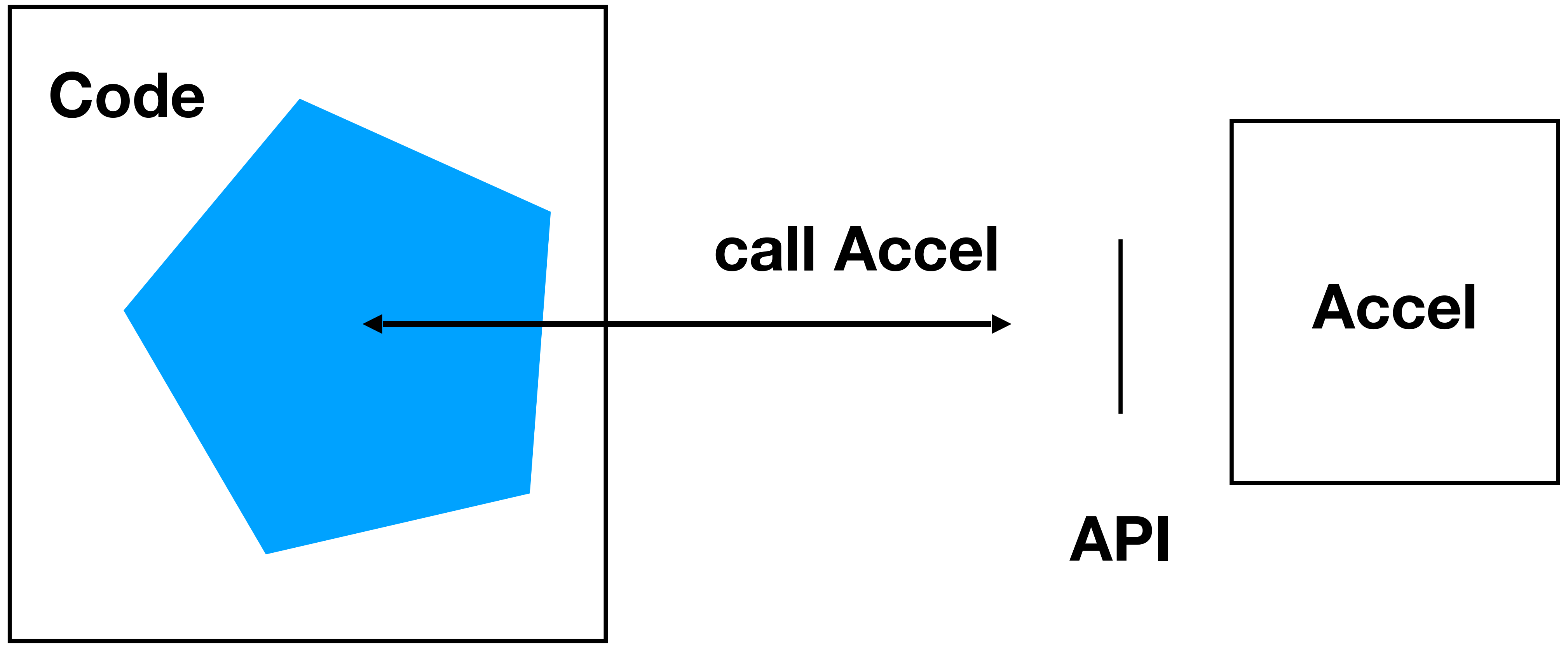
Neural Compilation [21-?]
- language to assembler translation using NMT/transformer

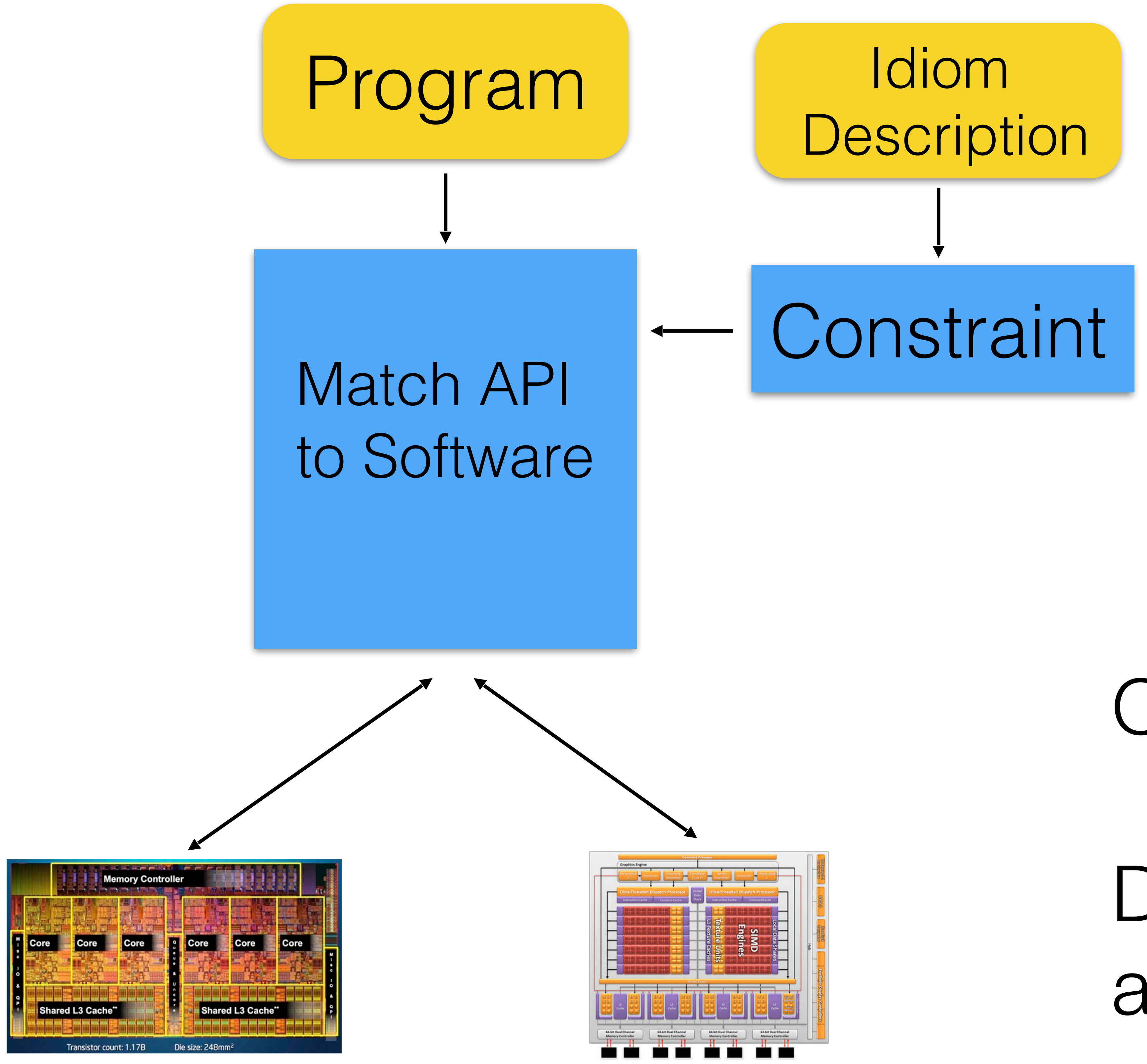
Program Lifting [22-?]
- beyond APIs lifting to DSLs/MLIR

Detect code structures that match interface



Detect code structures that match interface

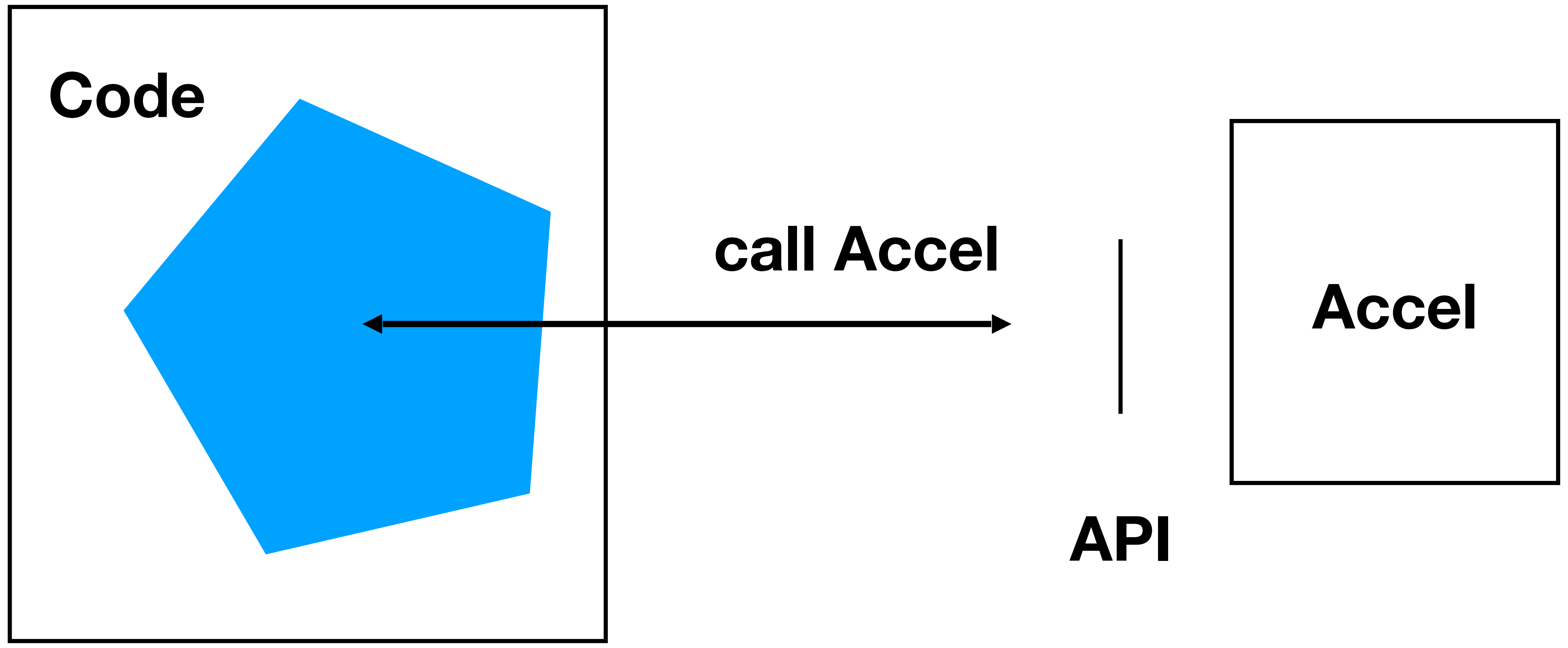


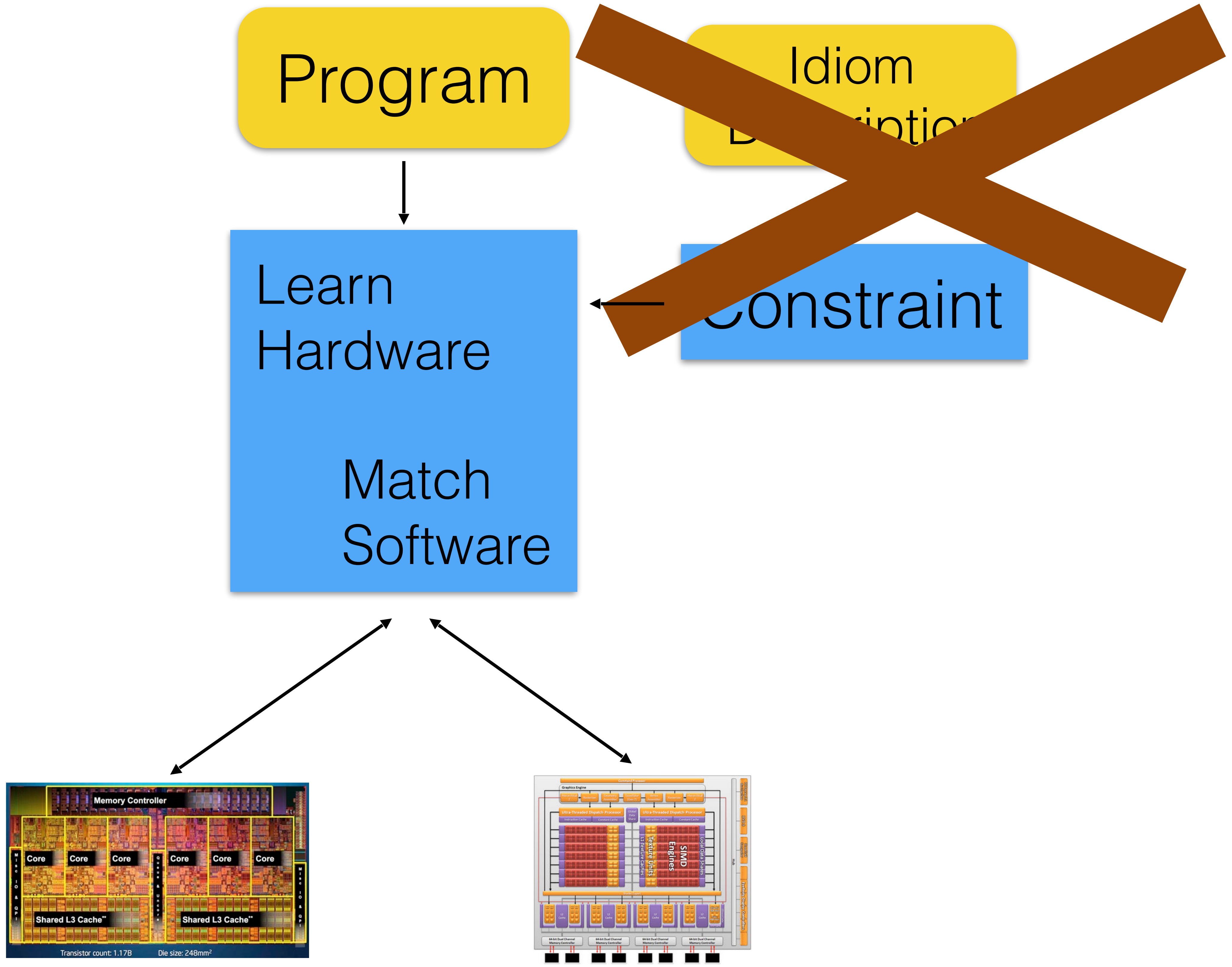


Challenge:

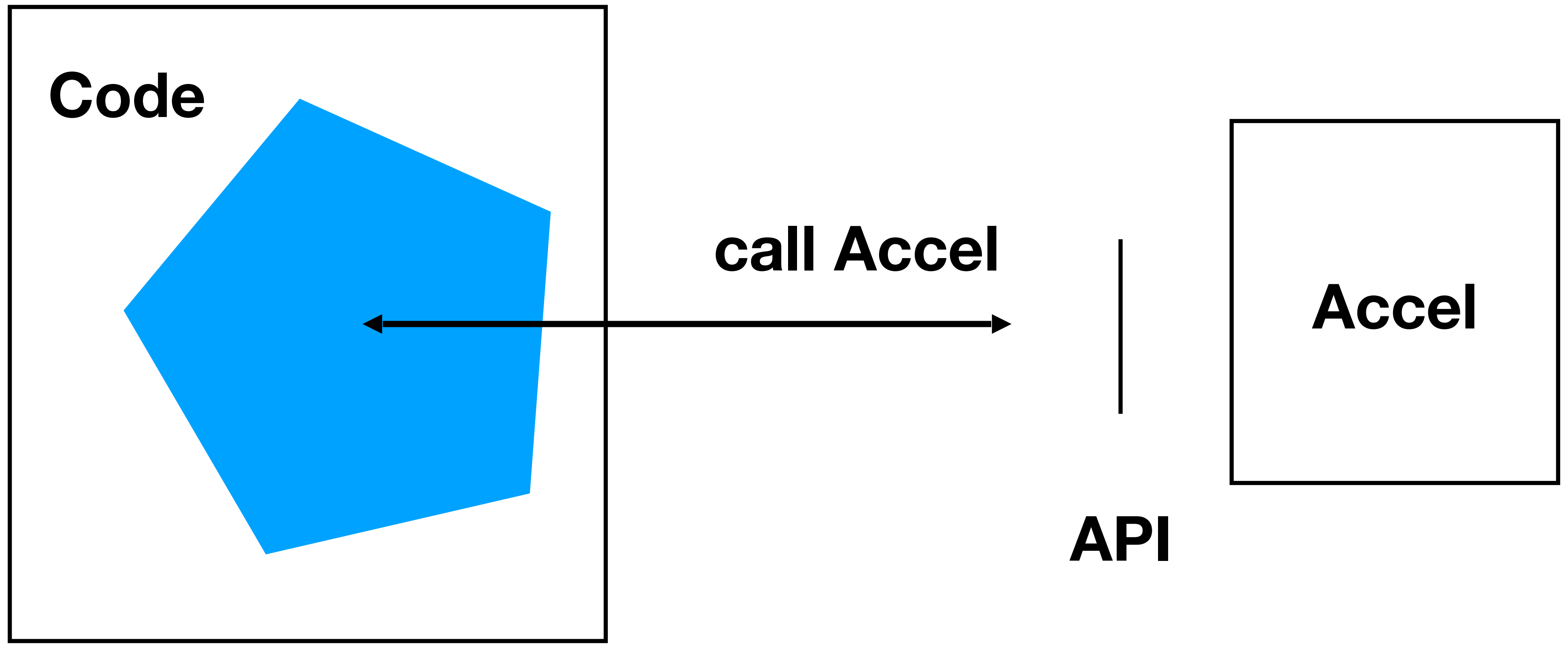
Do this entirely automatically

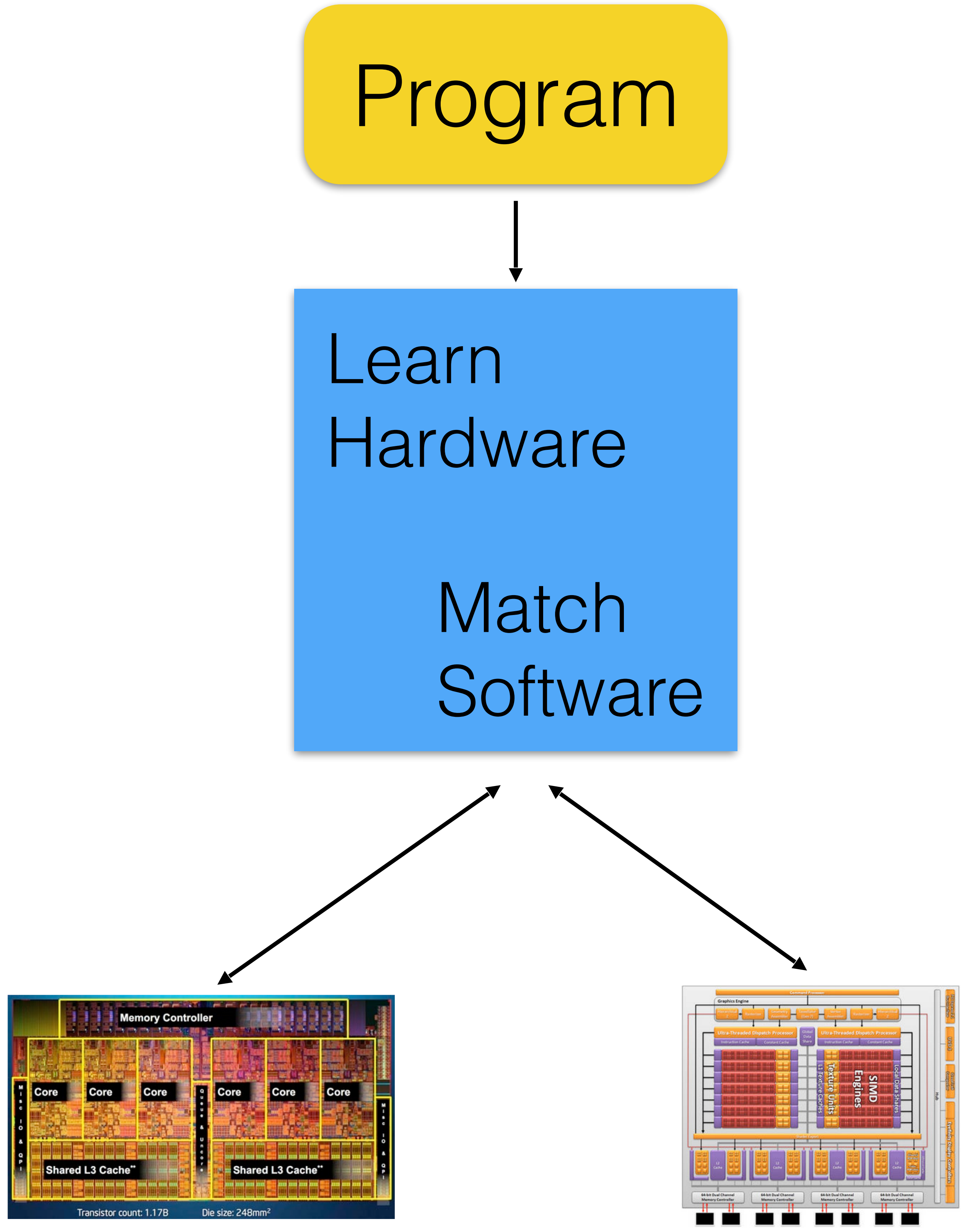
Detect code structures that match interface





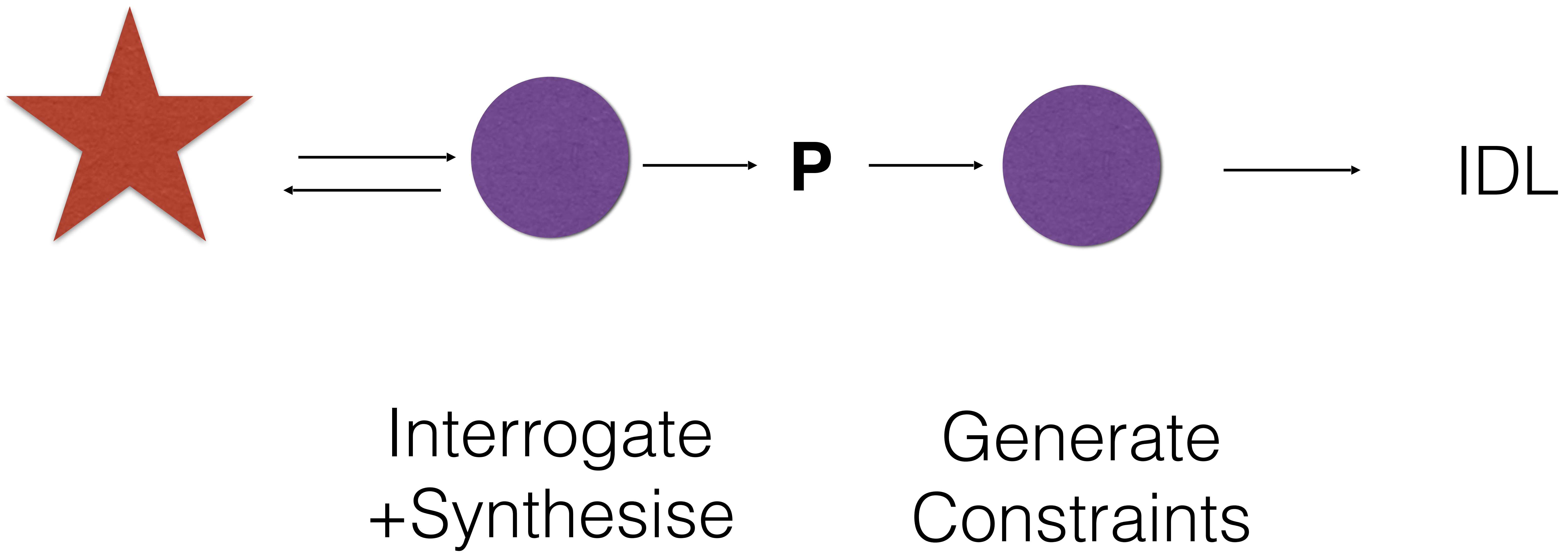
Detect and match automatically





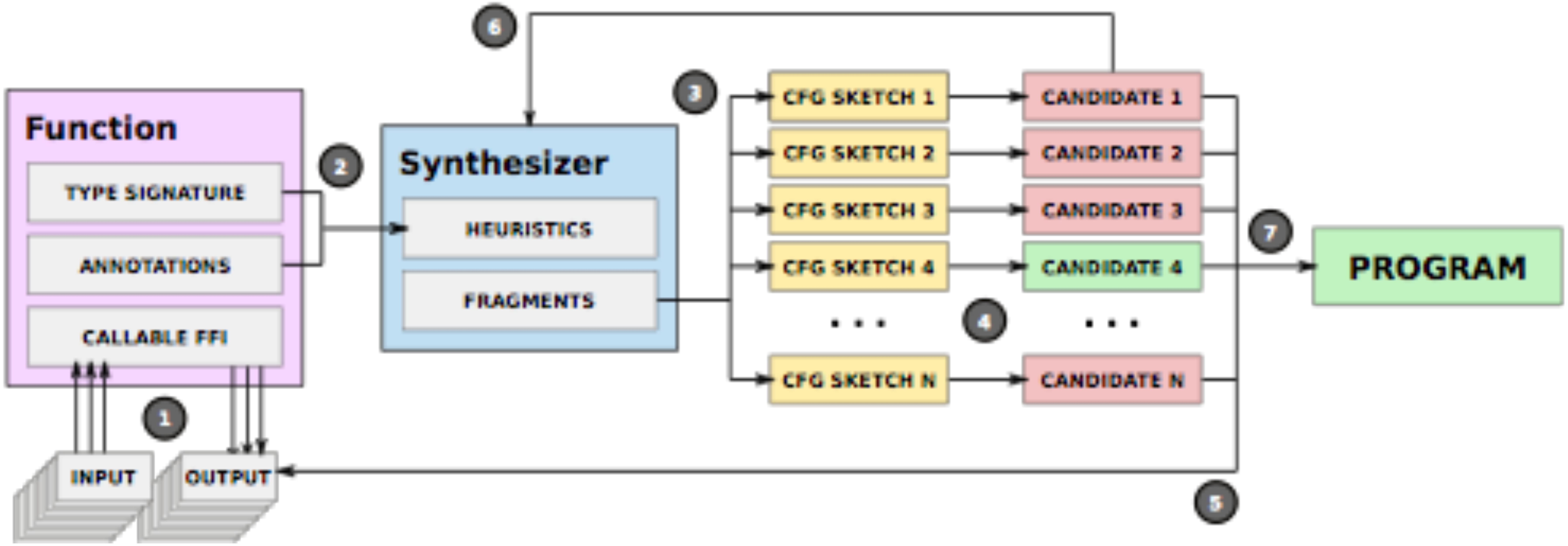
Use IO, probabilistic and grey-box program synthesis

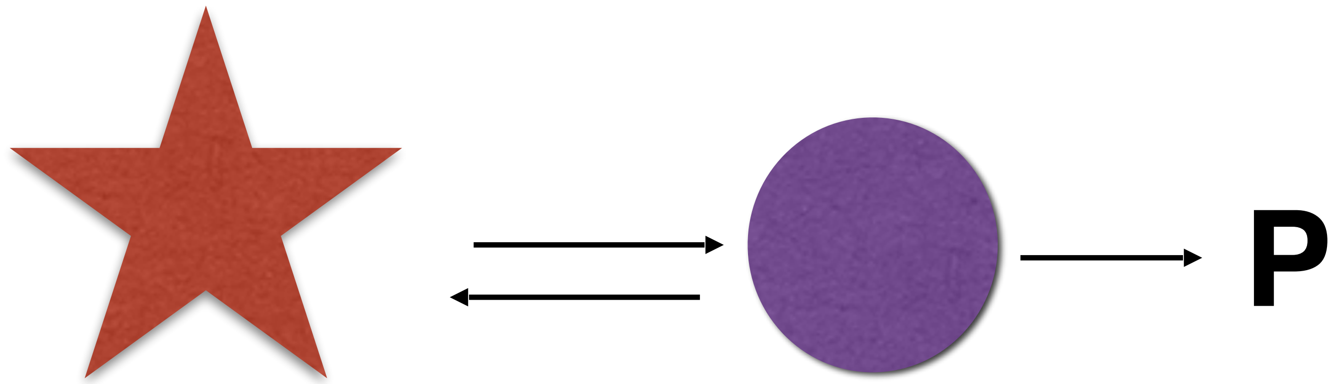
Auto-discovery: Synthesise +Generalise

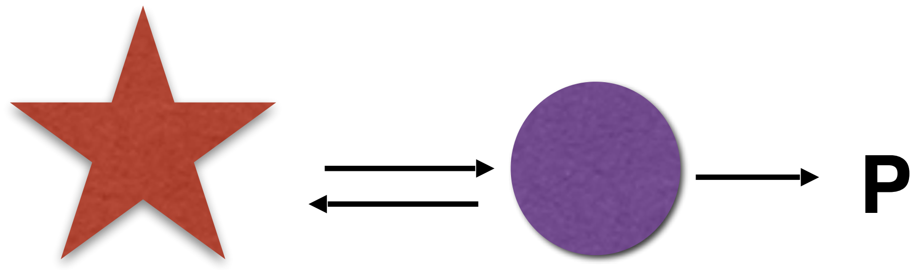


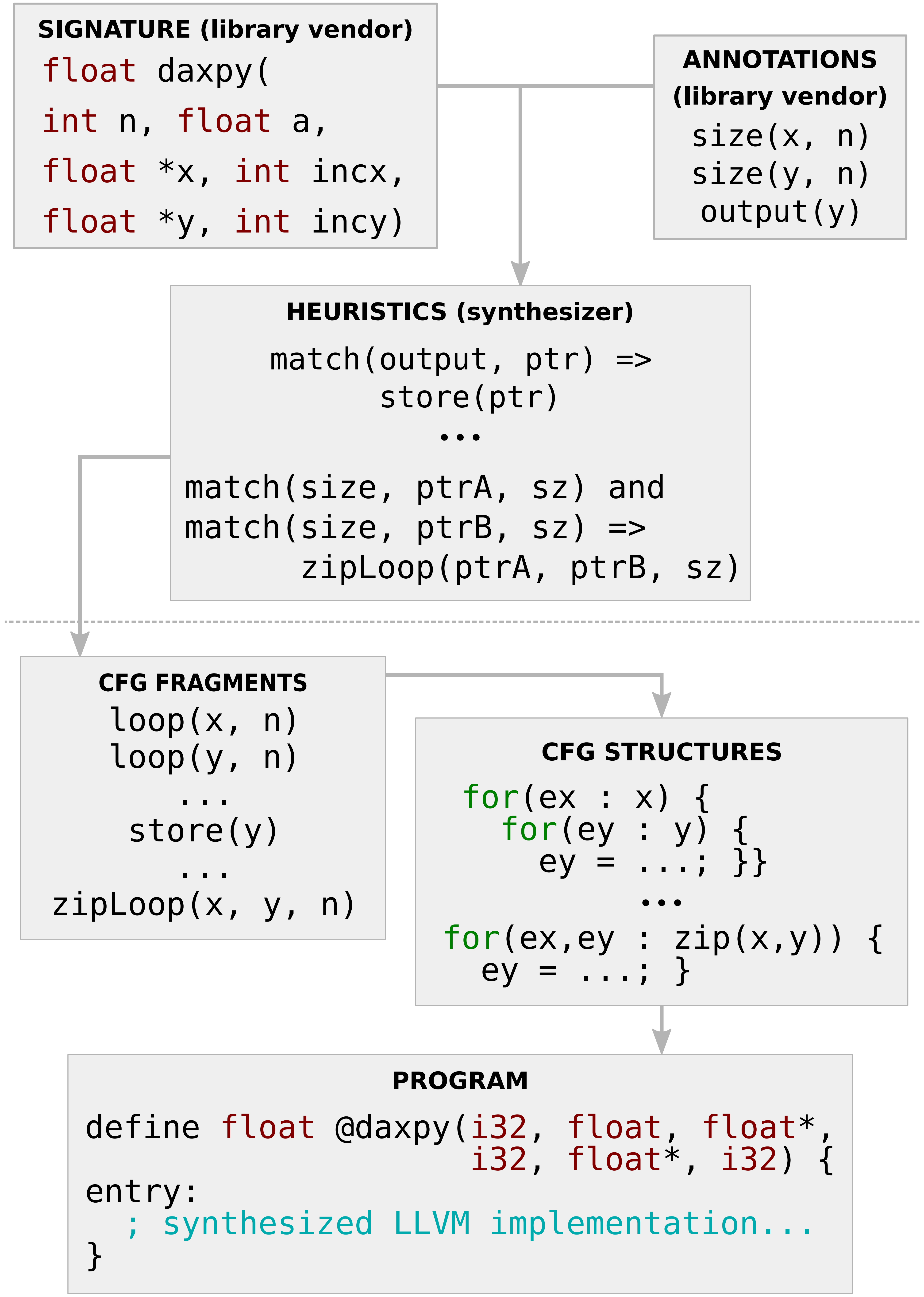
Constraint generation from program is trivial. How to generate a program P ?

Type directed synthesis

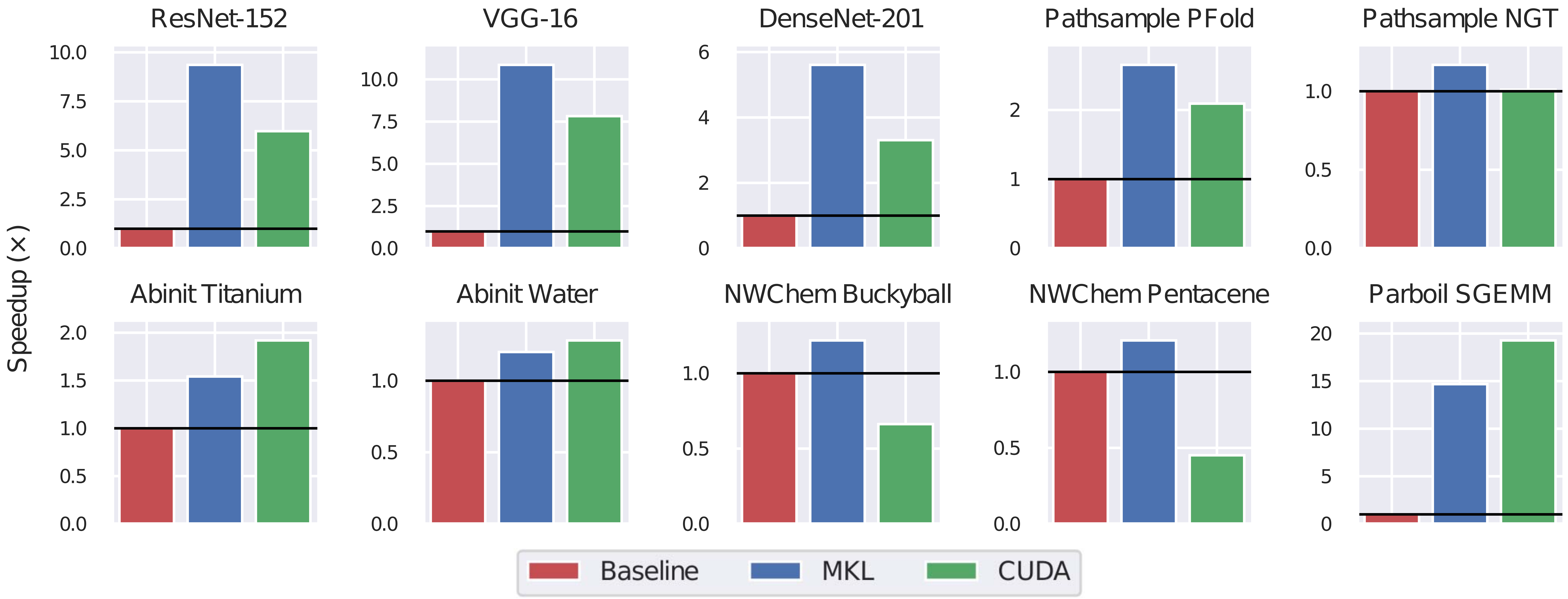





P

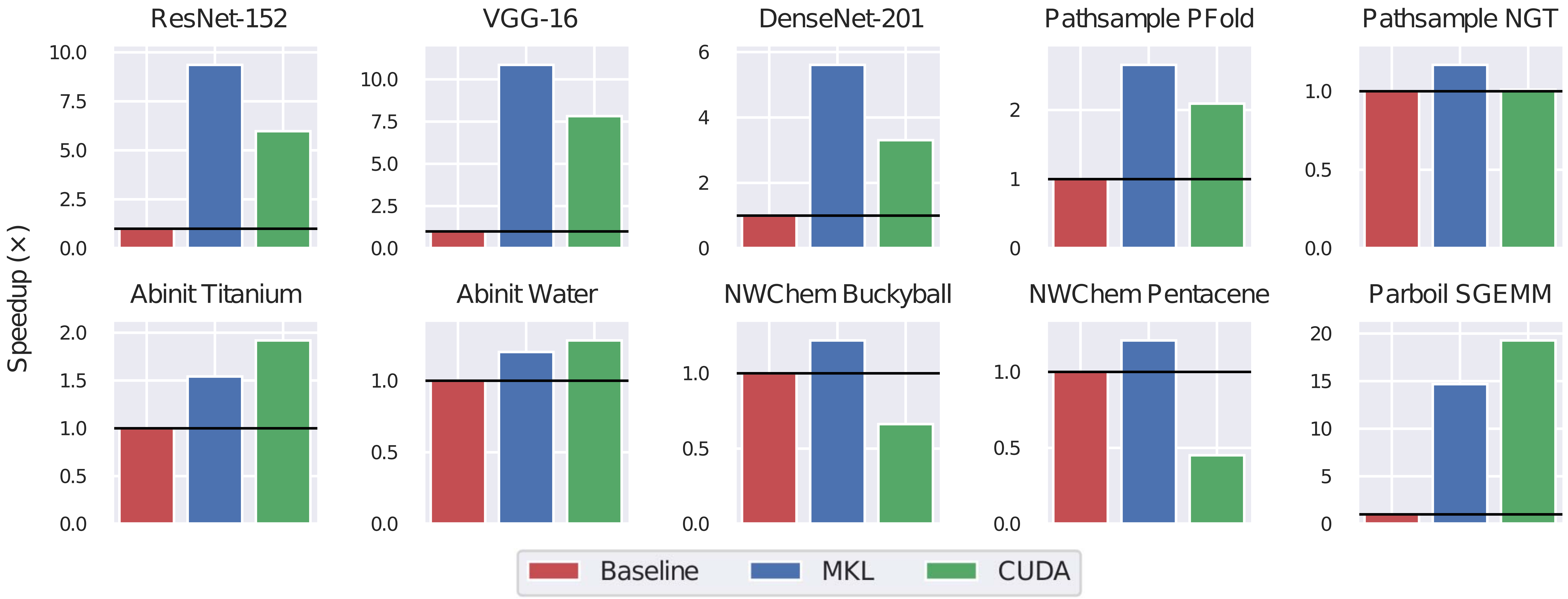

SIGNATURE (library vendor)
float daxpy(
int n, float a,
float *x, int incx,
float *y, int incy)
ANNOTATIONS
(library vendor)
size(x, n)
size(y, n)
output(y)
HEURISTICS (synthesizer)
match(output, ptr) =>
store(ptr)
...
match(size, ptrA, sz) and
match(size, ptrB, sz) =>
zipLoop(ptrA, ptrB, sz)
CFG FRAGMENTS
loop(x, n)
loop(y, n)
...
store(y)
...
zipLoop(x, y, n)
CFG STRUCTURES
for(ex : x) {
for(ey : y) {
ey = ...; }}
...
for(ex,ey : zip(x,y)) {
ey = ...; }
PROGRAM
define float @daxpy(i32, float, float*,
i32, float*, i32) {
entry:
; synthesized LLVM implementation...
}

It works



[PACT19]

It works



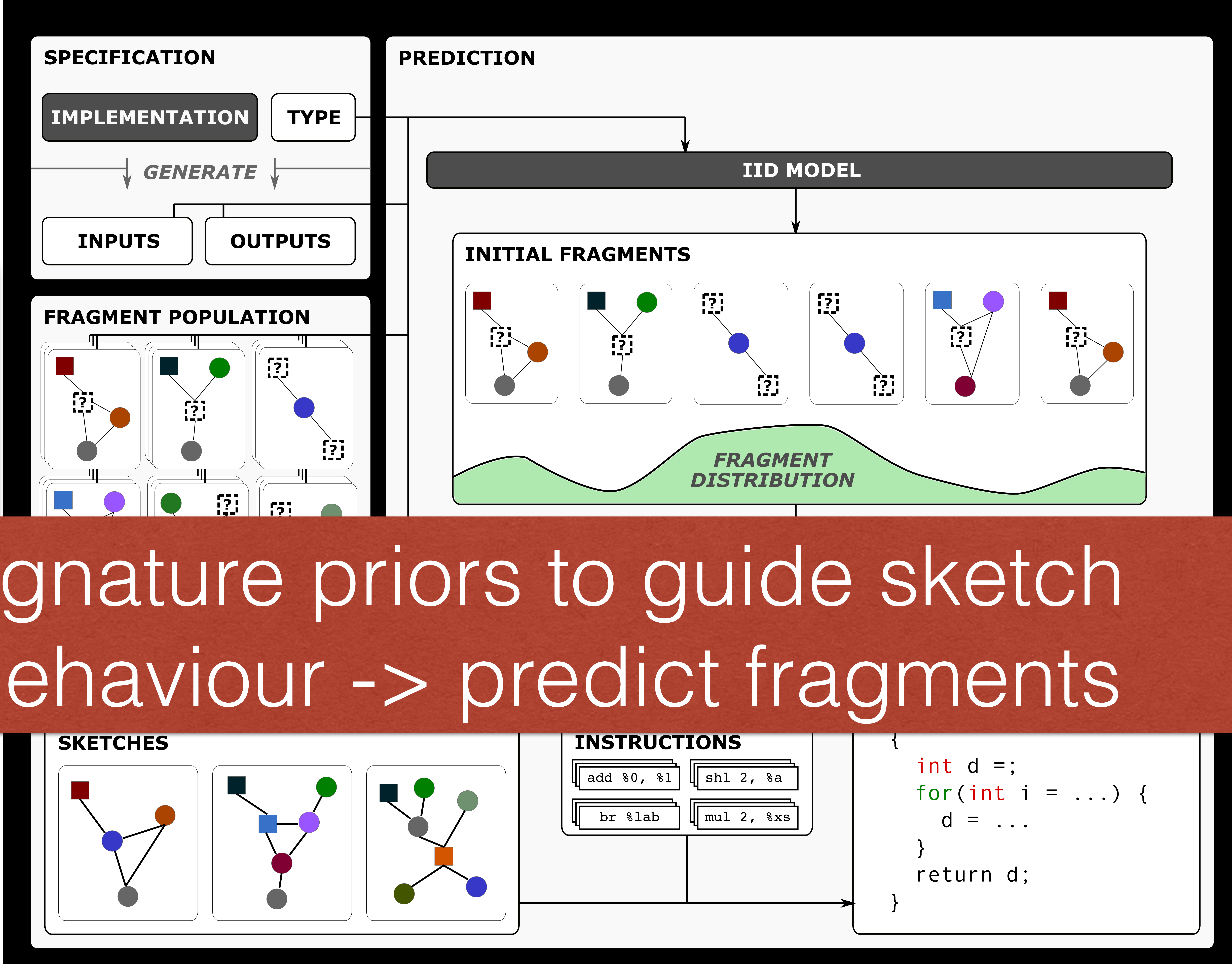
[PACT19] **But requires type annotations - not fully automatic**


INPUTS
Outputs
black_box(Inputs) {
// implementation...
}
OUTPUTS
SPECIFICATION
IMPLEMENTATION
TYPE
GENERATE
INPUTS
OUTPUTS
FRAGMENT POPULATION
PREDICTION
IID MODEL
INITIAL FRAGMENTS
FRAGMENT DISTRIBUTION
MARKOV MODEL
SYNTHESIS
ITERATE
SAMPLE
SKETCHES
INSTRUCTIONS
add %0, %1
shl 2, %a
br %lab
mul 2, %xs
PROGRAM
int f(int a, int *b)
{
int d =;
for(int i = ...) {
d = ...
}
return d;
}

INPUTS

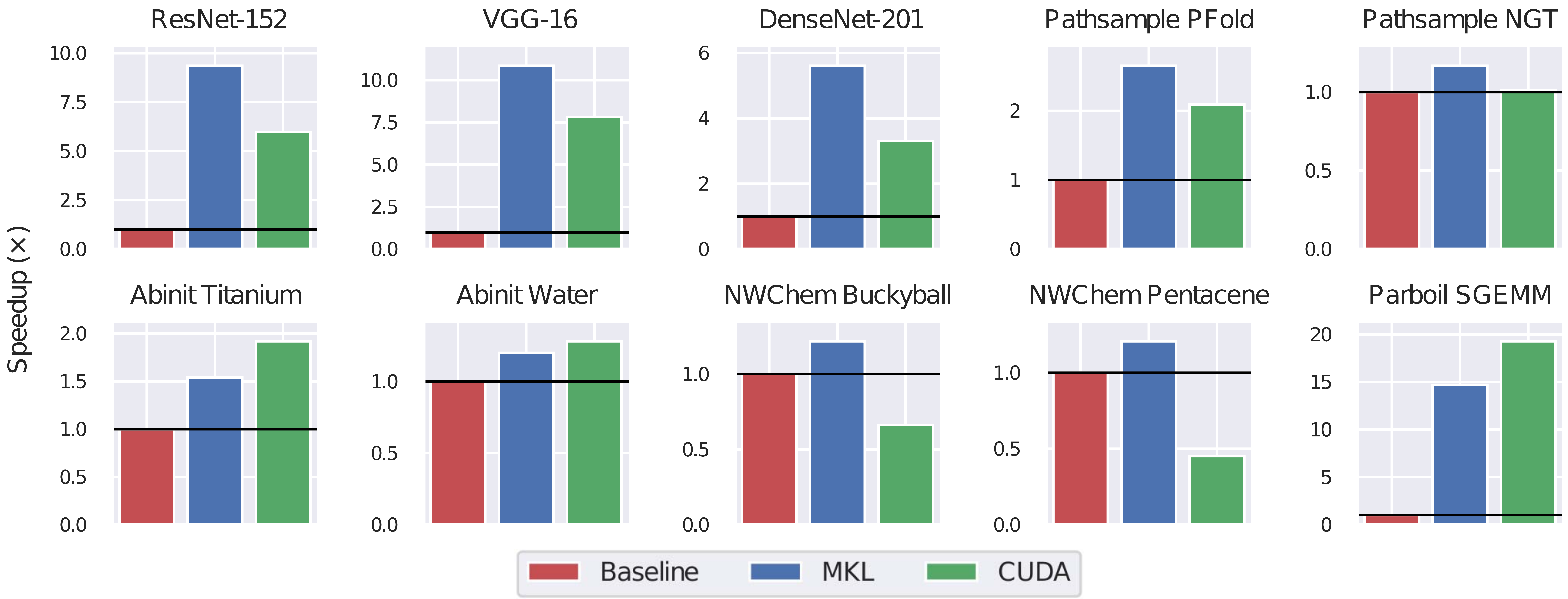
Outputs
black_box(Inputs) {

OUTPUTS



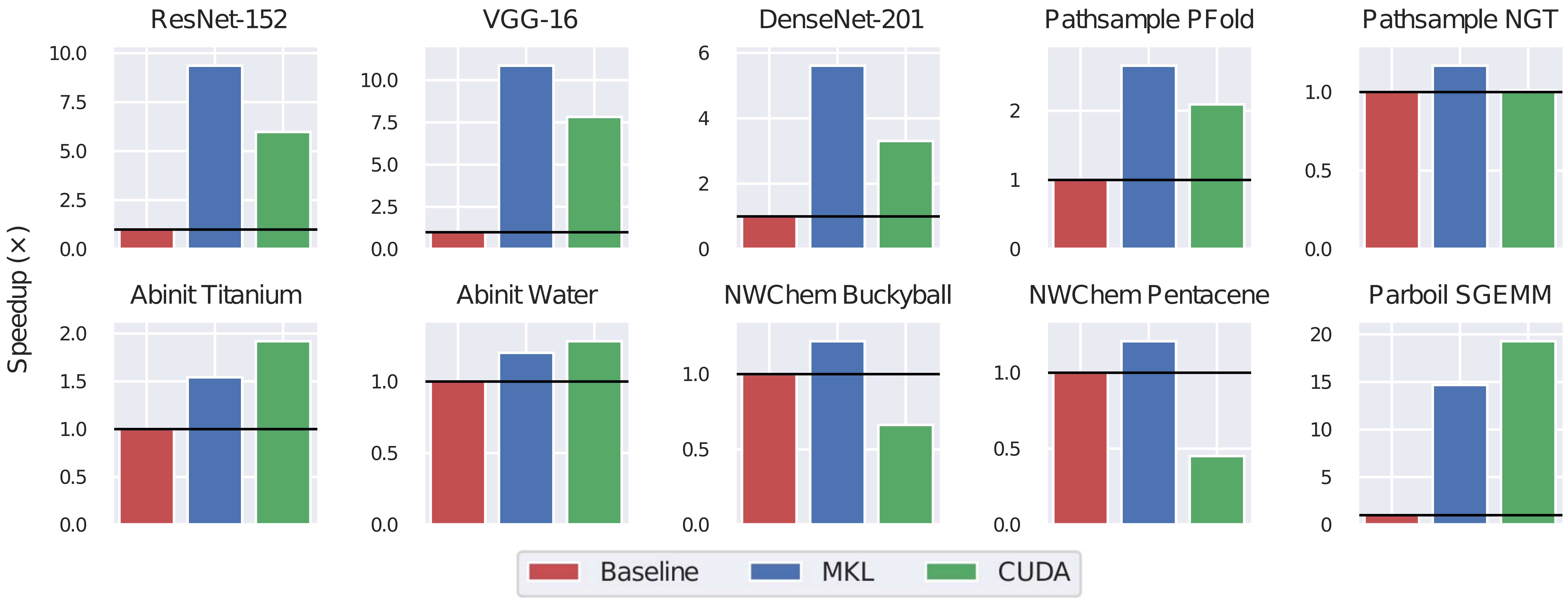
Use ML signature priors to guide sketch IO, grey behaviour -> predict fragments

It works



[PACT19]

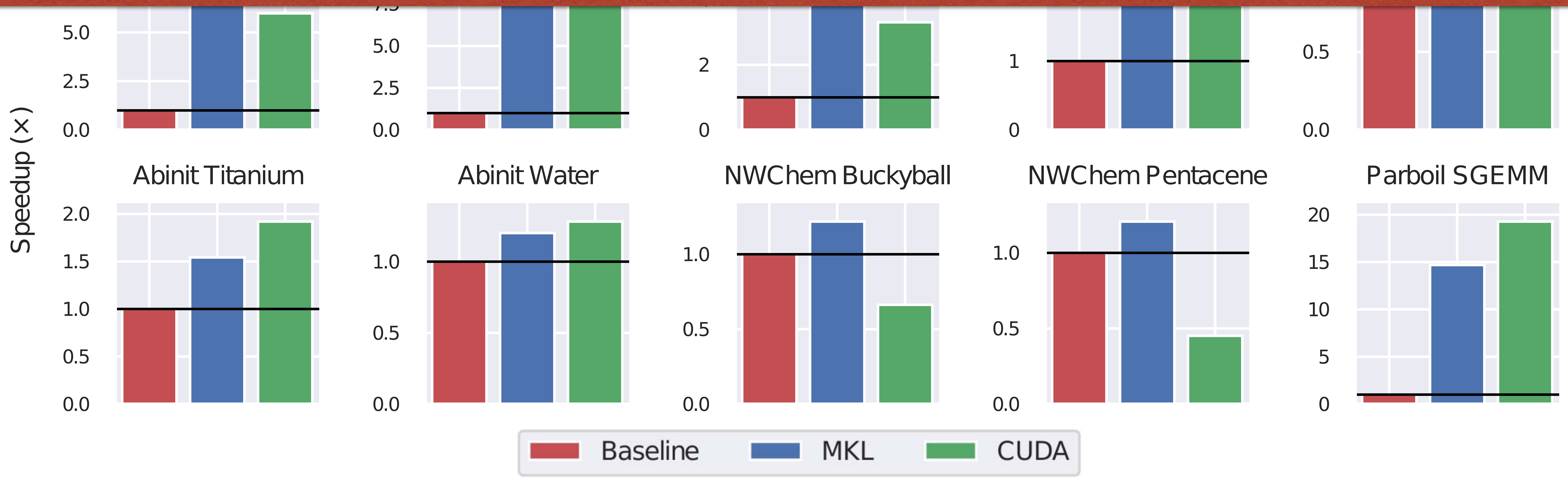
It really works!



[PACT19] [ASE20] [GPCE20] [PACT21]

Remove annotation hints, Use prior and grey knowledge

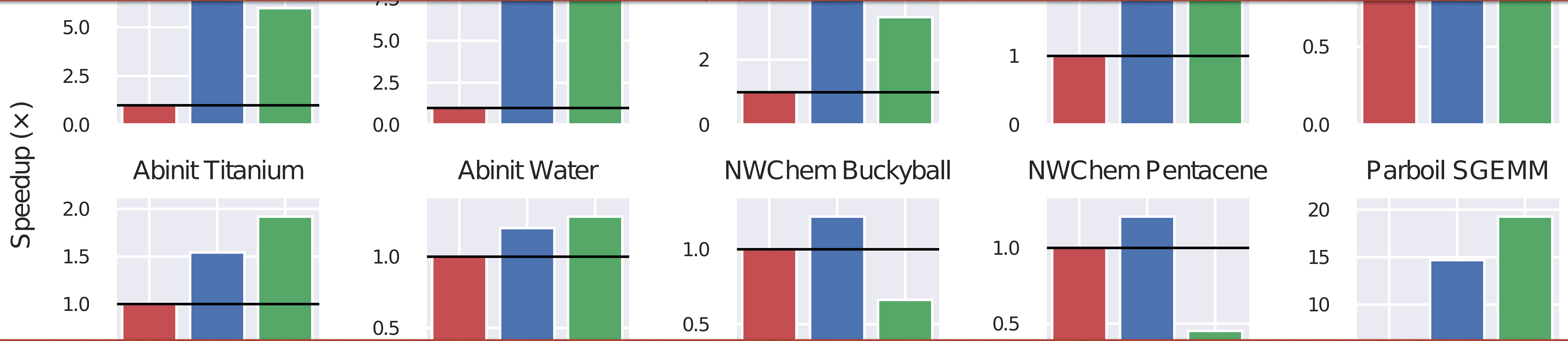
Automatically matches accelerator libraries to legacy code



[PACT19] [ASE20] [GPCE20] [PACT21]

Remove annotation hints, Use prior and grey knowledge

Automatically matches accelerator libraries to legacy code



No programmer in the loop

[PACT19] [ASE20] [GPCE20] [PACT21]

Remove annotation hints, Use prior and grey knowledge

5 approaches to lifting

Search using constraints over LLVM IR: IDL+CanDL [18-20]
- targetted APIs in C/Fortran - dense/sparse linear algebra

Black-box Program Synthesis [19-21]
- eliminated need for writing constraints

API matching via IO behavioural equivalence [21-23]
- more robust detection

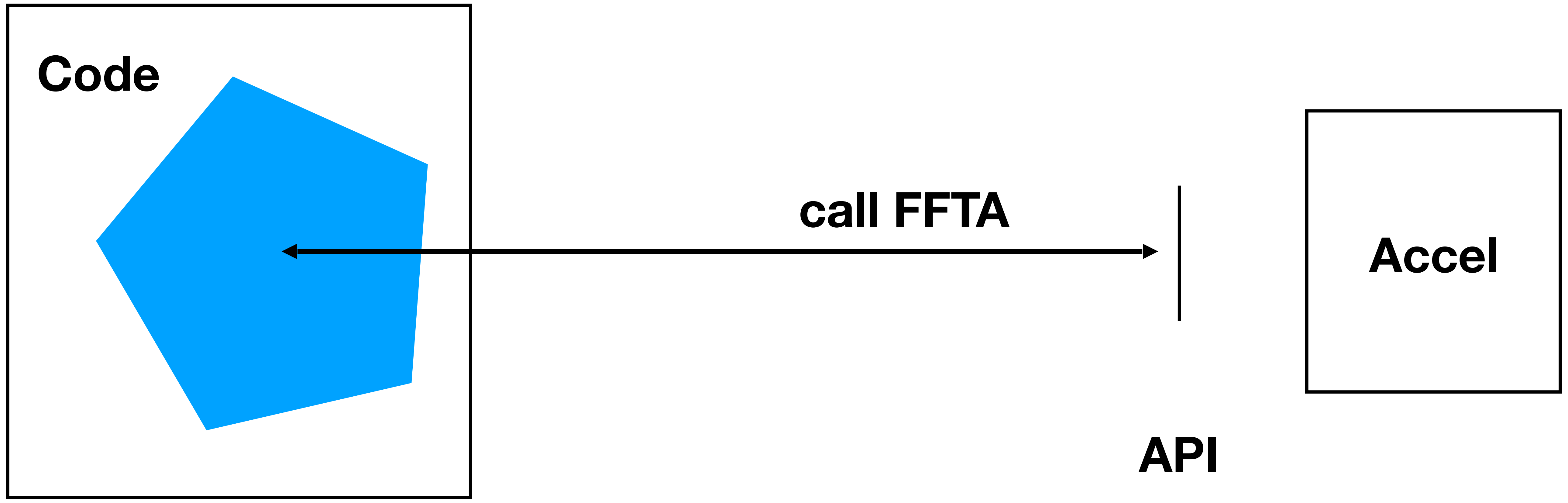
Neural Compilation [21-?]
- language to assembler translation using NMT/transformer

Program Lifting [22-?]
- beyond APIs lifting to DSLs/MLIR

Big-step Acceleration: FFT

Although accelerator *discovery* is possible

Matching complex accelerators to code
is challenging

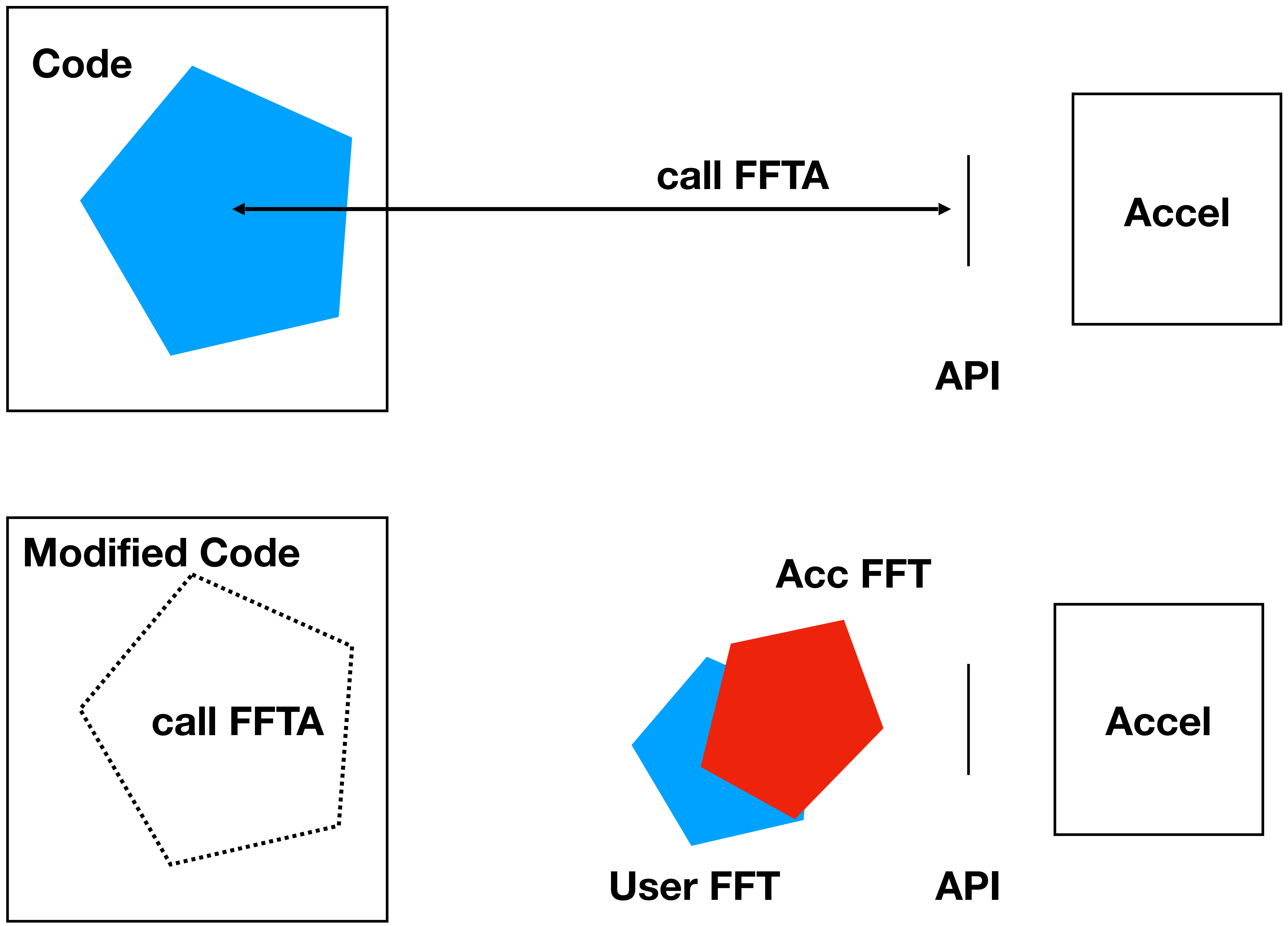


Big-step Acceleration: FFT

Matching complex accelerators is challenging

- Behaviour unlikely to match user code

- FFT acceleration a good example



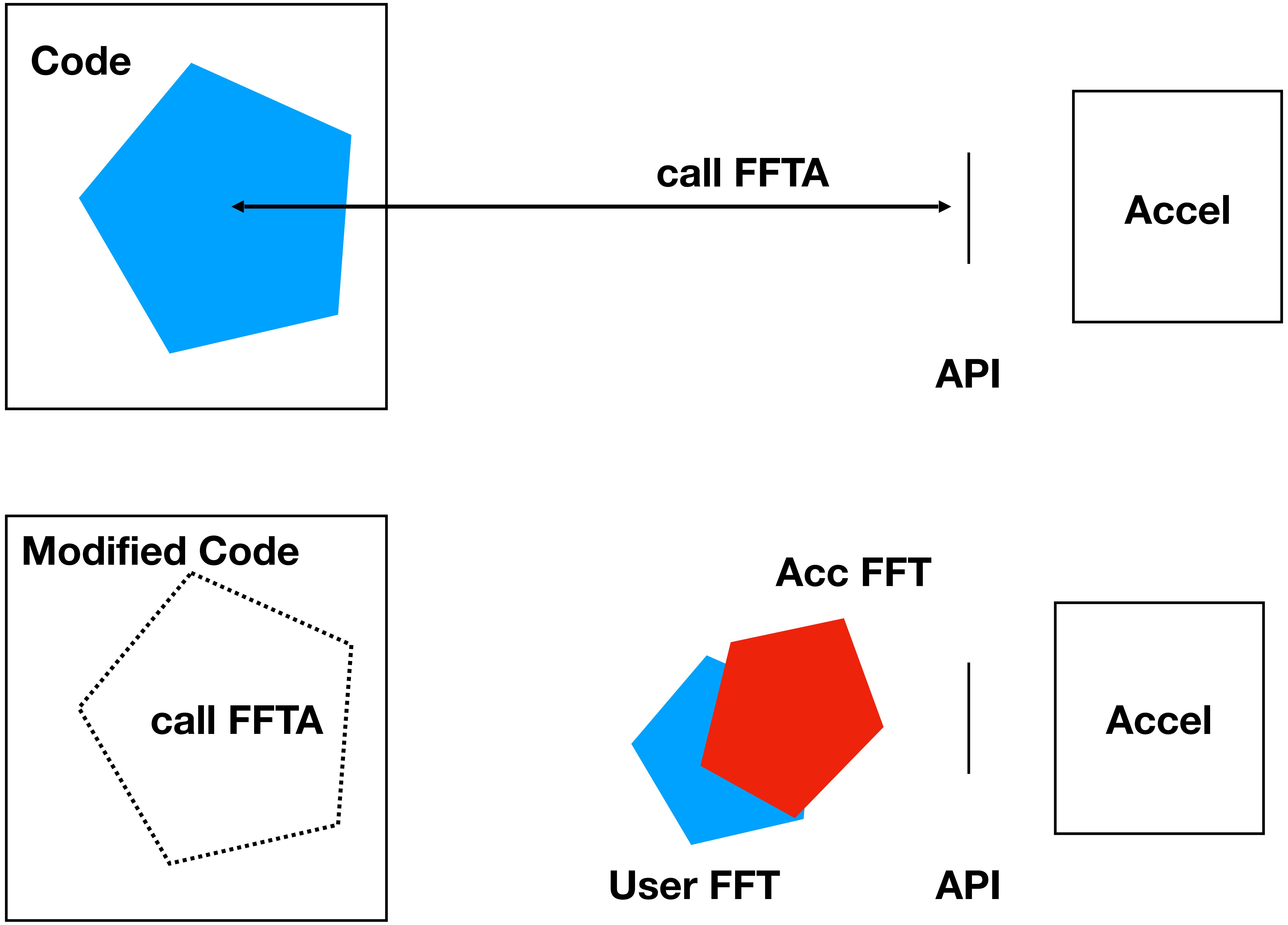
Bridge the gap on real code

Need to bridge gap

- Applied to Raw C GitHub code

- Discovered, modified and replaced
  - with libs or accelerators

- FFTW, SHARC DSP, PowerQuad



Neural Classifier + IO behaviour

Rather than constraints to match

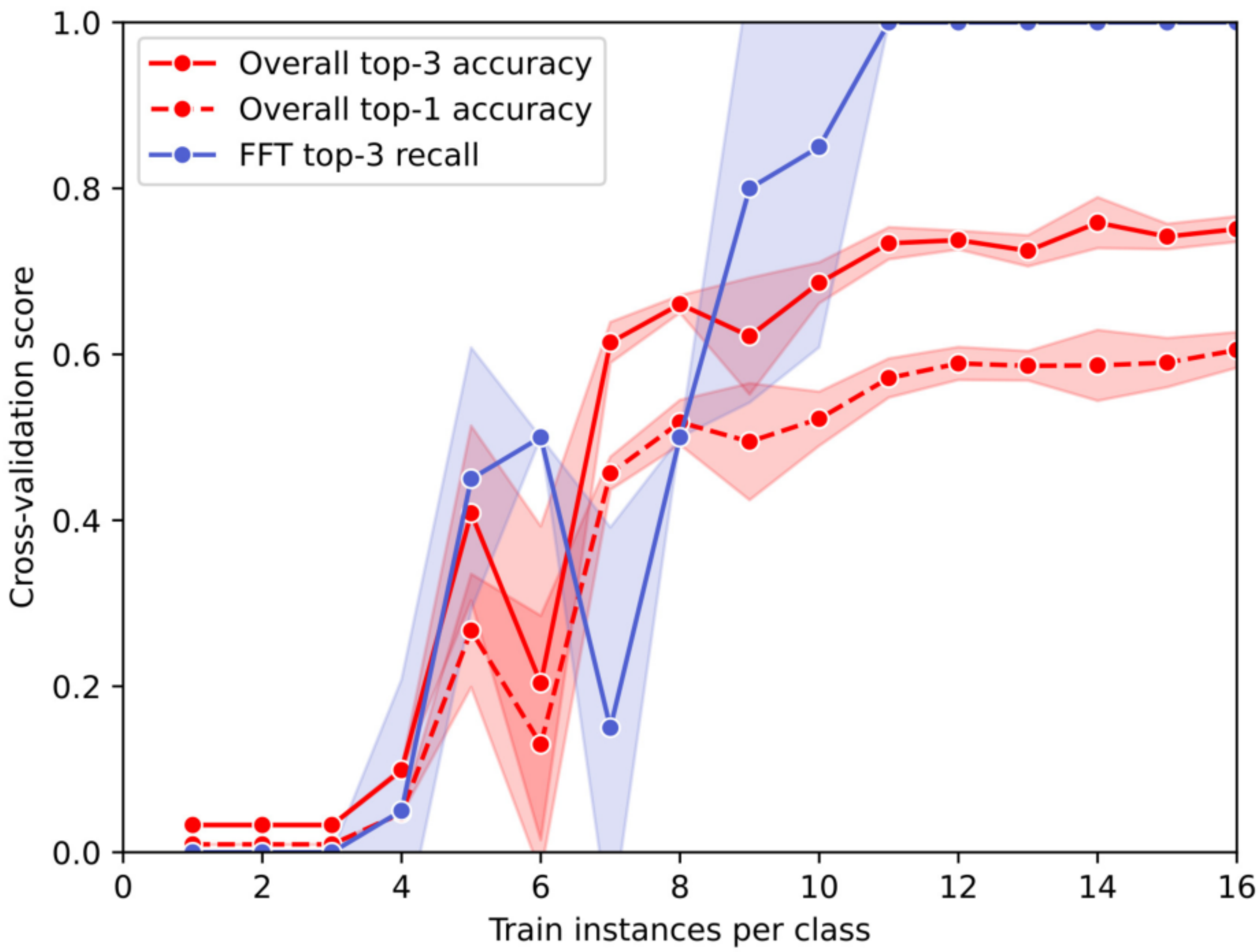
Use a **neural classifier**

- detects FFT***ish*** Github code

Then IO behavioural equivalence

- does it have same behaviour?

Patch up with specialised synthesised normalisation code



[PLDI22]

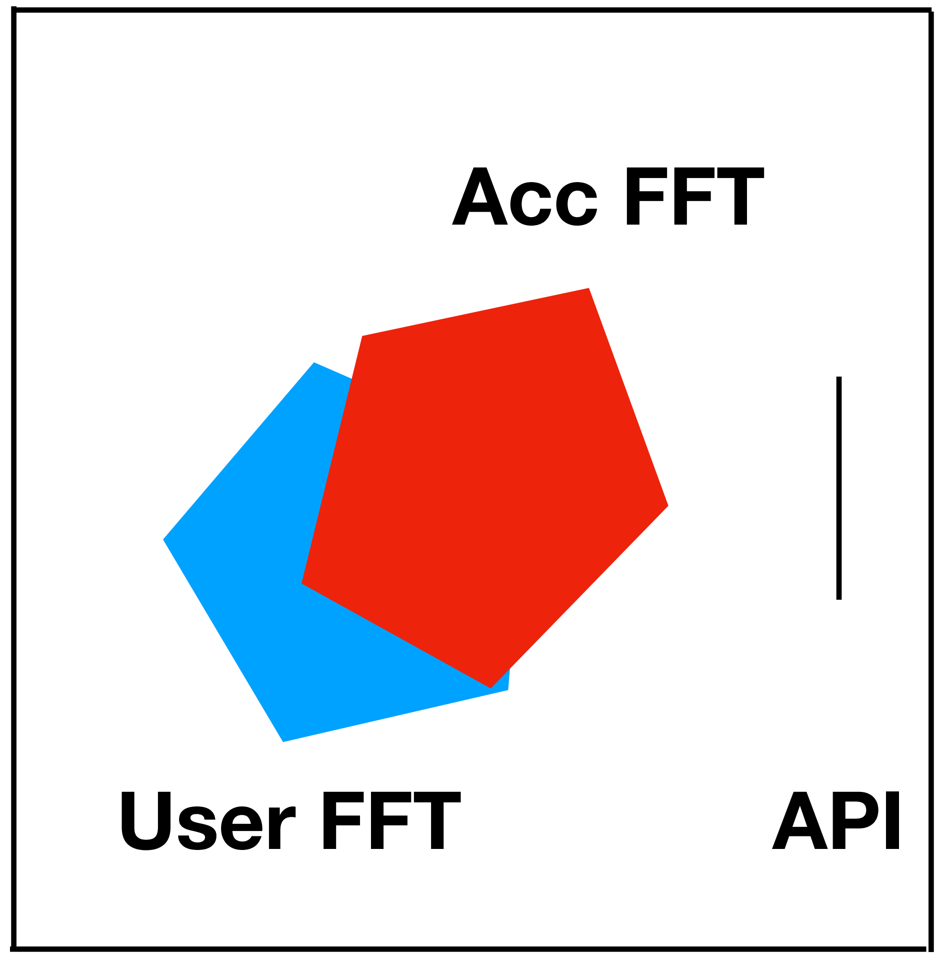
Big-step Acceleration: FFT

Project	Lines of Code	Lengths Supported	Algorithm	Twiddle Factors	Imaginary Numbers	Pointer Arithmetic	Loop Structure	Optimizations
0	83	Only 64	Radix-2 FFT	Constant	Custom	No	While-True-Break	Minimal
1	278	Powers of 2 (≤ 256)	Radix-2 FFT	Constant	Custom	No	Do-While/For	Minimal
2	65	Powers of 2	Radix-2 FFT	Computed in FFT	Custom	No	For/Recursive	Minimal
3	107	Powers of 2	Radix-2 FFT	Computed in FFT	Custom	No	For	Minimal
4	934	All	Mixed-Radix FFT	Computed in FFT	Custom	No	For/Recursive	Extensive Unrolling
5	2159	All	Mixed-Radix FFT	Pre-Computed	Custom	Yes	For	Hand-Vectorized/Unrolled
6	77	Powers of 2	Radix-2 FFT	Computed in FFT	Custom	No	For	Minimal
7	237	Powers of 2	Radix-2 FFT	Pre-Computed	Custom	Yes	For	Minimal
8	101	Powers of 2	Radix-2 FFT (DIF)	Computed in FFT	C99 Complex	No	For	Minimal
9	1627	All	Mixed-Radix FFT	Pre-Computed	Custom	Yes	For/While/Recursive	Extensive Unrolling
10	75	Powers of 2	Radix-2 FFT	Pre-Computed	Custom	No	For	Minimal
11	538	All	Mixed-Radix FFT	Pre-Computed	Custom	Yes	Do-While/For	Twiddle-Factor Memoization
12	367	All	Mixed-Radix + Bluestein	Computed in FFT	Custom	No	For/Recursive	Unrolling
13	101	Powers of 2	Radix-2 FFT (DIT)	Computed in FFT	C99 Complex	No	For	Minimal
14	314	Powers of 2	Radix-2 FFT	Computed in FFT	None	No	For	Minimal
15	215	All	Recursive FFT	Computed in FFT	C99 Complex	No	Recursive	Minimal
16	20	All	DFT	Unneeded	C99 Complex	No	For	None
17	12	All	DFT	Unneeded	C99 Complex	No	For	None

GitHub code in the wild: Vast range of styles, quality, behaviour

Big-step Acceleration: FFT

Automatically generates adaptor code




```
complex *FFT_accel(complex *x, int N) {
  // Check for valid inputs to accelerator
  if (is_power_of_two(N) && N <= 65536) {
    // Bind user inputs to accelerator
    int len = N;
    #pragma align 64
    complex_float output[len];
    complex_float input[len];
    #pragma end
    for (int i = 0; i < len; i++) {
      input[i].re = x[i].real;
      input[i].im = x[i].imag;
    }
    // Call accelerator
    accel_cfft(input, output, len);
    // Bind accelerator outputs
    for (int j = 0; j < N; j++) {
      x[j].imag = output[j].im;
      x[j].real = output[j].re;
    }
    // De-normalize outputs
    for (int k = 0; k < N; i ++)  {
      x[k].imag *= N;
      x[k].real *= N;
    }
  } else { // Not valid accelerator input
    // Fallback to user code.
    UserFFT(x, N);
  }
}
```

Big-step Acceleration: FFT

Automatically generates adaptor code

- Range check

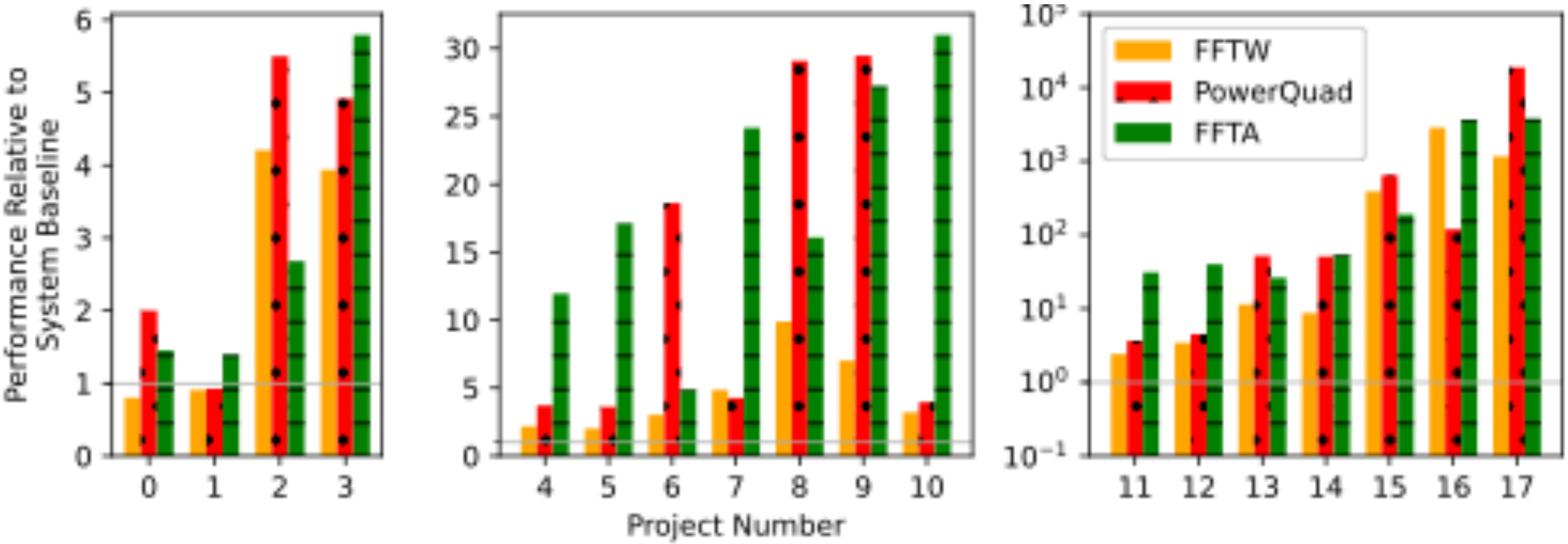
- Type conversion

- Variable binding

- Sythesized normalisation code

```
complex *FFT_accel(complex *x, int N) {
  // Check for valid inputs to accelerator
  if (is_power_of_two(N) && N <= 65536) {
    // Bind user inputs to accelerator
    int len = N;
    #pragma align 64
    complex_float output[len];
    complex_float input[len];
    #pragma end
    for (int i = 0; i < len; i++) {
      input[i].re = x[i].real;
      input[i].im = x[i].imag;
    }
    // Call accelerator
    accel_cfft(input, output, len);
    // Bind accelerator outputs
    for (int j = 0; j < N; j++) {
      x[j].imag = output[j].im;
      x[j].real = output[j].re;
    }
    // De-normalize outputs
    for (int k = 0; k < N; i ++)  {
      x[k].imag *= N;
      x[k].real *= N;
    }
  } else { // Not valid accelerator input
    // Fallback to user code.
    UserFFT(x, N);
  }
}
```

Big-step Acceleration: FFT



Speedup over CPU baseline using either FFTW library, FFTA Sharc DSP, NXP PowerQuad

Project numbers refer to legacy C GitHub code

Applied to Github linear algebra

Algorithm	Code	LoC	Layout	Sizes	Optimizations
Naive	1	11	Column-major	Squared	None
	2	117	Both	Any	None
	3	15	Row-major	Any	None
	4	23	Column-major	Squared	None
	5	27	Row-major	Squared	OpenMP
	6	9	Row-major	Any	None
	7	9	Row-major	Any	None
	8	18	Column-major	Squared	OpenMP
	9	131	Row-major	Any	OpenMP
	10	12	Row-major	Any	None
	11	18	Row-major	Multiple of nthreads	C++ threads
	12	63	Row-major	Squared	C++ threads
	13	16	Column-major	Any	None
	14	31	Column-major	Any	None
	15	31	Column-major	Any	None
	16	38	Row-major	Any	None
	17	8	Row-major	Squared	None
Unrolled	18	43	Row-major	Any	None
	19	38	Row-major	Any	None
	20	43	Row-major	Squared	OpenMP
	21	33	Row-major	Squared, multiple of bs	None
Kernel Calls	22	23	Column-major	Any	None
	23	89	Column-major	Any	OpenMP
	24	26	Column-major	Any	None
	25	62	Column-major	Any	Unrolled

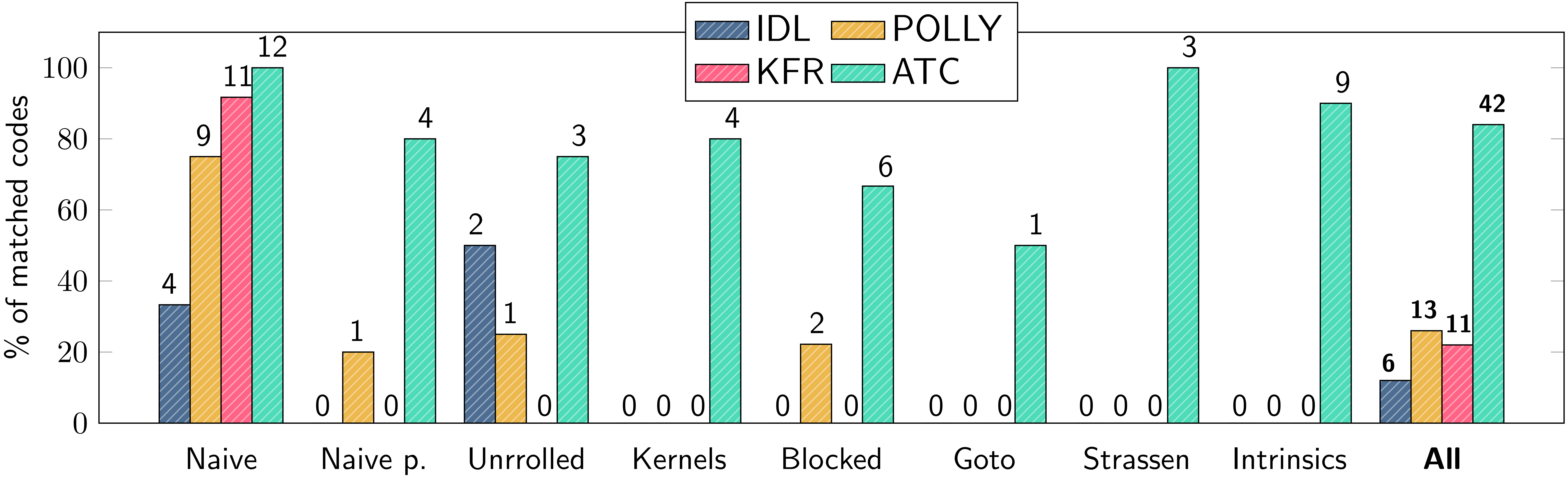
Algorithm	Code	LoC	Layout	Sizes	Optimizations
Kernel Calls	26	106	Column-major	Any	Unrolled
Blocked	27	76	Row-major	Any	Block
	28	21	Row-major	Squared	OpenMP
	29	41	Column-major	Any	None
	30	31	Row-major	Squared	None
	31	27	Column-major	Squared	None
	32	37	Row-major	Multiple of bs	Unrolled
	33	44	Row-major	Squared	None
	34	13	Row-major	Squared	None
	35	16	Row-major	Squared	None
Goto	36	176	Column-major	Squared	Intrinsics (SSE)
	37	54	Row-major	Squared	None
Strassen	38	152	Row-major	Squared	None
	39	200	Row-major	Squared, power of 2	None
	40	82	Row-major	Squared	None
Intrinsics	41	75	Row-major	Squared	Intrinsics (AVX2)
	42	76	Row-major	Multiple of 8	Intrinsics (AVX2)
	43	62	Row-major	Multiple of 8	Intrinsics (AVX2)
	44	53	Row-major	Any	Intrinsics (SSE)
	45	89	Row-major	Multiple of bs	Intrinsics (AVX2)
	46	108	Row-major	Multiple of bs	Intrinsics (AVX2)
	47	287	Row-major	Any	Intrinsics (AVX2)
	48	354	Row-major	Multiple of bs	Intrinsics (AVX2)
	49	44	Row-major	Multiple of bs	Intrinsics (AVX2)
	50	62	Row-major	Any	Intrinsics (SSE)

Strassen and intrinsics

```
// P0 = A*(F - H);
msub(n, Ypitch, F, Ypitch, H, n, T);
mmult_fast(n, Xpitch, A, n, T, n, P[0]);

// P1 = (A + B)*H
madd(n, Xpitch, A, Xpitch, B, n, T);
mmult_fast(n, n, T, Ypitch, H, n, P[1]);

// P2 = (C + D)*E
madd(n, Xpitch, C, Xpitch, D, n, T);
mmult_fast(n, n, T, Ypitch, E, n, P[2]);
...

// Z upper left = (P3 + P4) + (P5 - P1)
madd(n, n, P[4], n, P[3], n, T);
msub(n, n, P[5], n, P[1], n, U);
madd(n, n, T, n, U, Zpitch, Z);

// Z lower left = P2 + P3
madd(n, n, P[2], n, P[3], Zpitch, Z + n*Zpitch);

// Z upper right = P0 + P1
madd(n, n, P[0], n, P[1], Zpitch, Z + n);

// Z lower right = (P0 + P4) - (P2 + P6)
madd(n, n, P[0], n, P[4], n, T);
madd(n, n, P[2], n, P[6], n, U);
msub(n, n, T, n, U, Zpitch, Z + n*(Zpitch + 1));
```

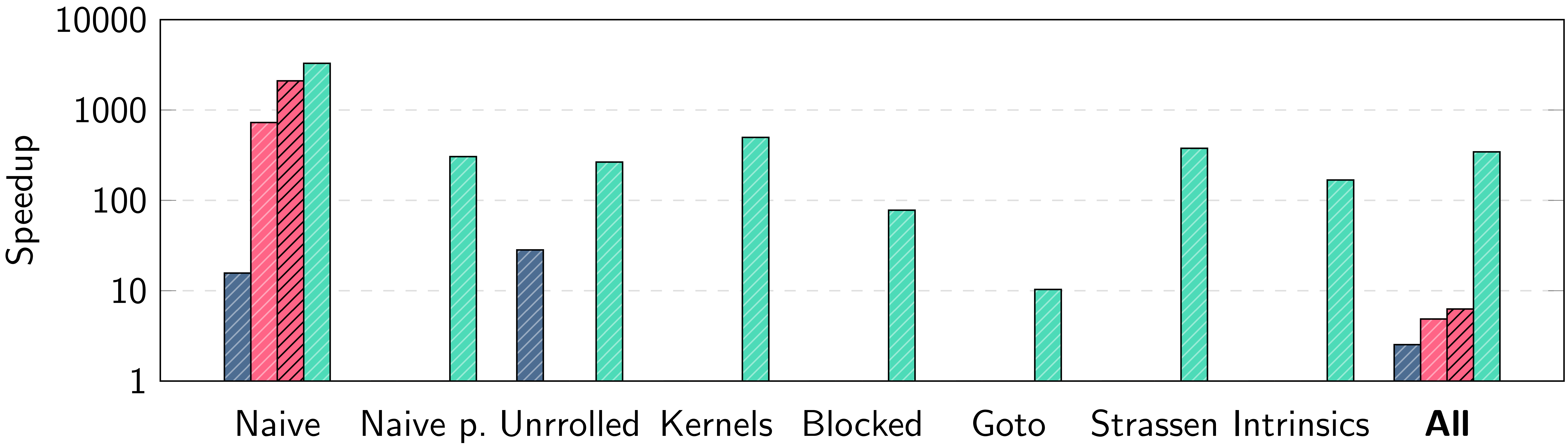
```
__m256 vab00 = _mm256_setzero_ps();
__m256 vab01 = _mm256_setzero_ps();
...

for (int k = 0; k < K; k++) {
  float pa = &A[lda * (k + i) + 0];
  float pb = &B[ldb * (k + i) + 0];

  __m256 vb0 = _mm256_load_ps(pb + 8 * 0);
  __m256 vb1 = _mm256_load_ps(pb + 8 * 1);

  __m256 va0 = _mm256_broadcast_ss(&pa[8 * i + 0]);
  __m256 va1 = _mm256_broadcast_ss(&pa[8 * i + 1]);
  ...

  vab00 = _mm256_fmadd_ps(va0, vb0, vab00);
  vab01 = _mm256_fmadd_ps(va0, vb1, vab01);
  ...
}

__m256 vc00 = _mm256_load_ps(C + ldc * 0 + 8 * 0);
...
vc00 = _mm256_add_ps(vc00, vab00);
...
_mm256_store_ps(C + ldc * 0 + 8 * 0, vc00);
```

GitHub code in the wild: Also applied to tensor convolutions



- Detects GEMMs in over 80% of cases

- Dramatic improvement over other approaches



- Leads to significant performance improvement

- Tensor cores on NVIDIA

- Similar results for convolutions on Google TPUs

[CC23]

Well known things

My view

Concrete results

Can we go further ?

Summary

5 approaches to lifting

Search using constraints over LLVM IR: IDL+CanDL [18-20]
- targetted APIs in C/Fortran - dense/sparse linear algebra

Black-box Program Synthesis [19-21]
- eliminated need for writing constraints

API matching via IO behavioural equivalence [21-23]
- more robust detection

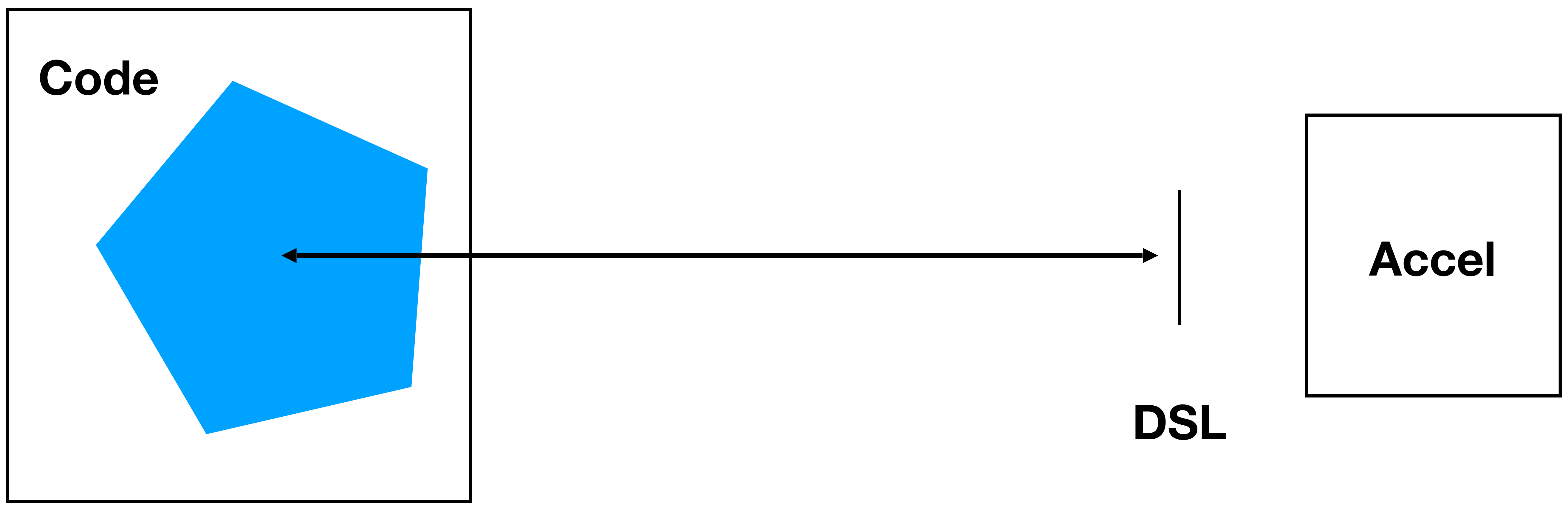
Neural Compilation [21-?]
- language to assembler translation using NMT/transformer

Program Lifting [22-?]
- beyond APIs lifting to DSLs/MLIR

Beyond fixed function: Neural Compilation

Significant accelerators will be programmable

- Likely to have specialised prog lang



Beyond fixed function: Neural Compilation

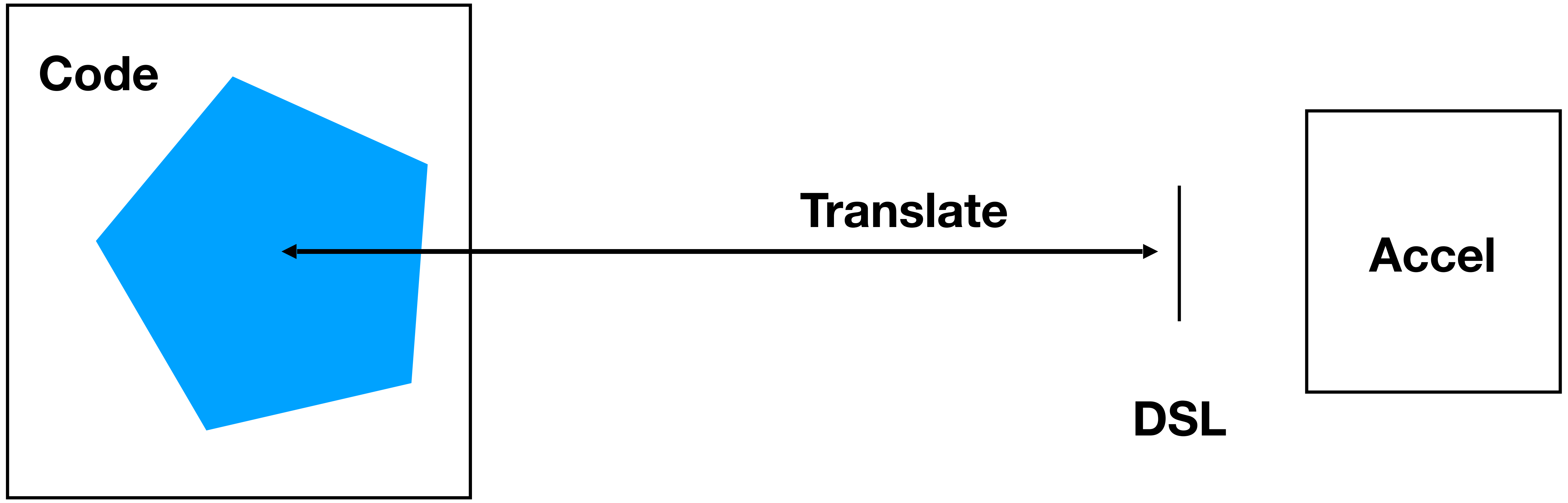
Significant accelerators will be programmable

- Likely to have specialised prog lang

Can we learn how to translate existing code into any new lang?

 - Automating compiler translation, construction

If so - enable language and architecture innovation



Beyond fixed function: Neural Compilation

Exploit advances in NLP
- Neuro Machine Translation (NMT)

NMT: Transformer model
- supervised translation of natural languages

NMT can perform **unsupervised** translation
- ie automatically translate between existing languages

Beyond fixed function: Neural Compilation

Exploit advances in NLP
- Neuro Machine Translation (NMT)

NMT: Transformer model
- supervised translation of natural languages

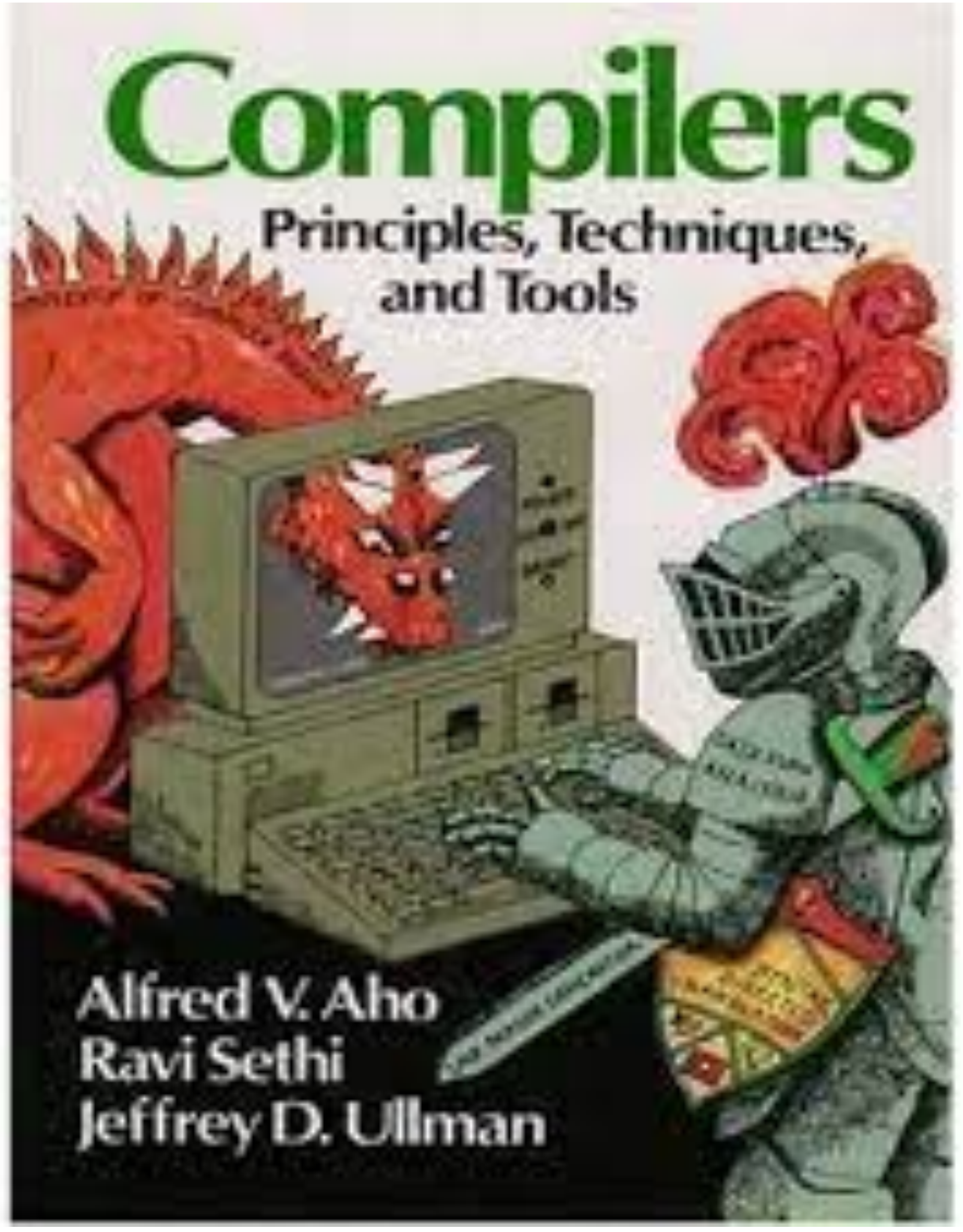
NMT can perform **unsupervised** translation
- ie automatically translate between existing languages

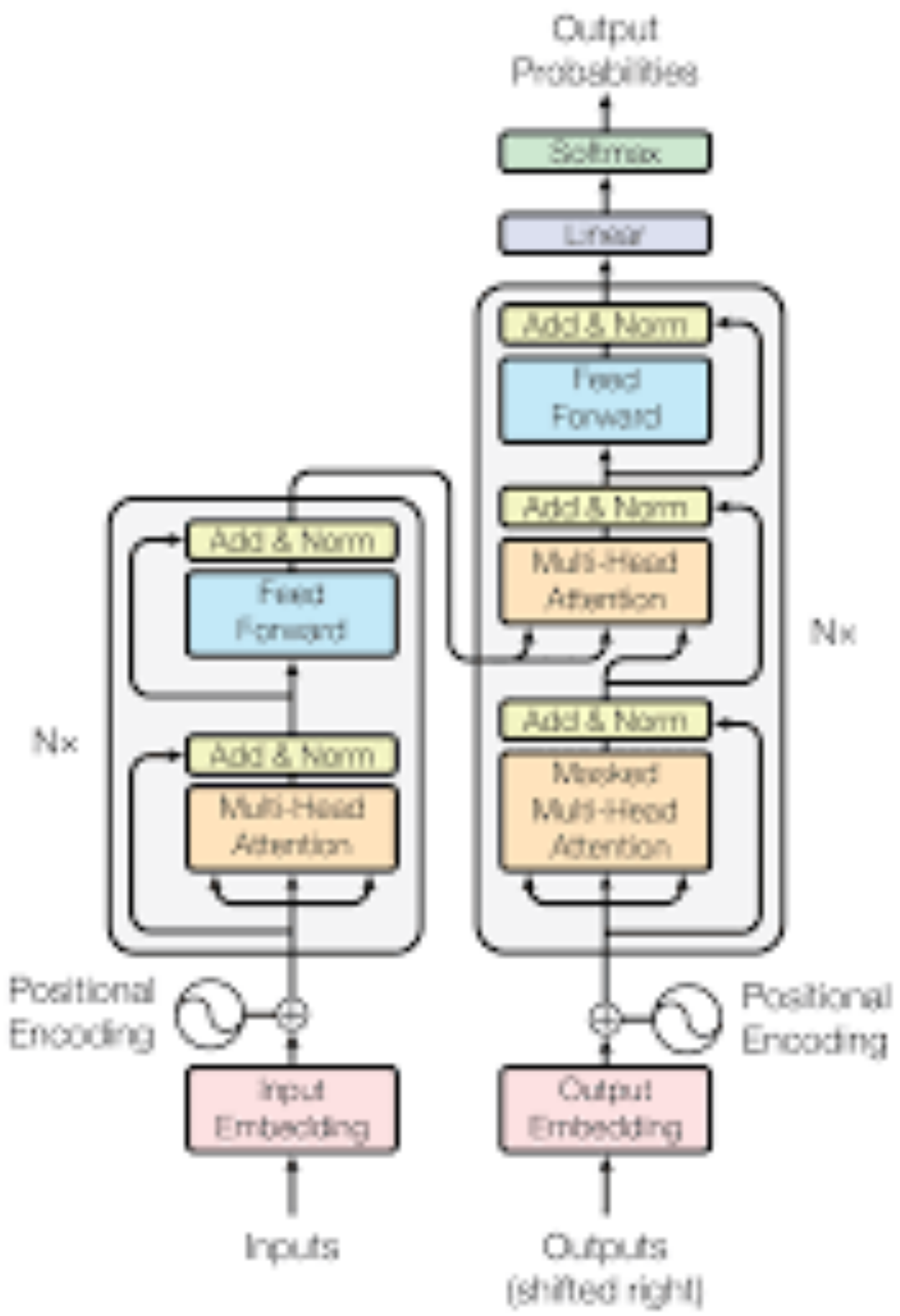
Can we do this for programming languages?

If so - potentially automate construction of compilers between any two languages

Let’s start with something “easy” supervised C->x86 compilation

Neural Compilation: C->x86 challenges





Neural Compilation: C->x86 challenges

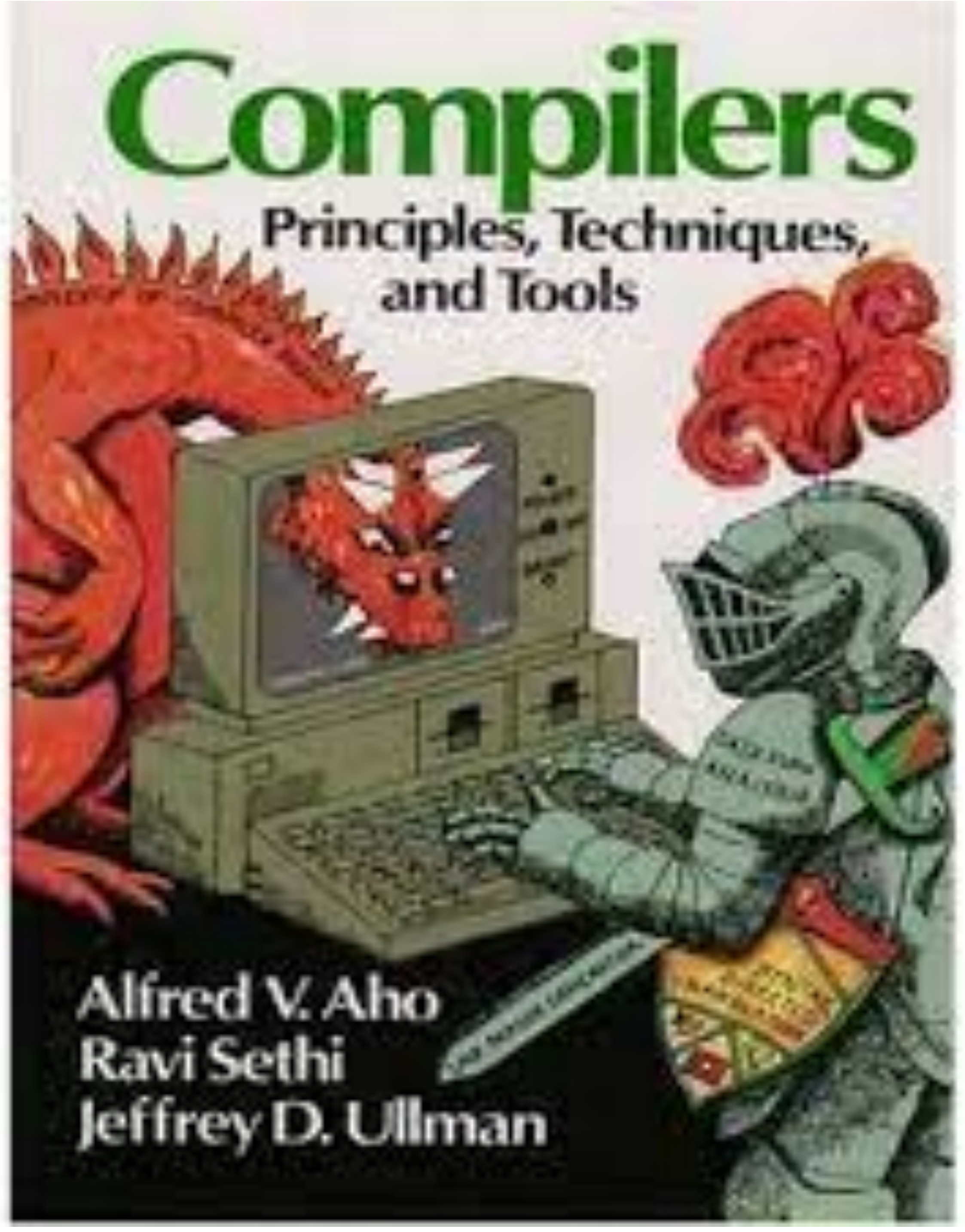
Exact solutions are needed
- nearly correct un-acceptable

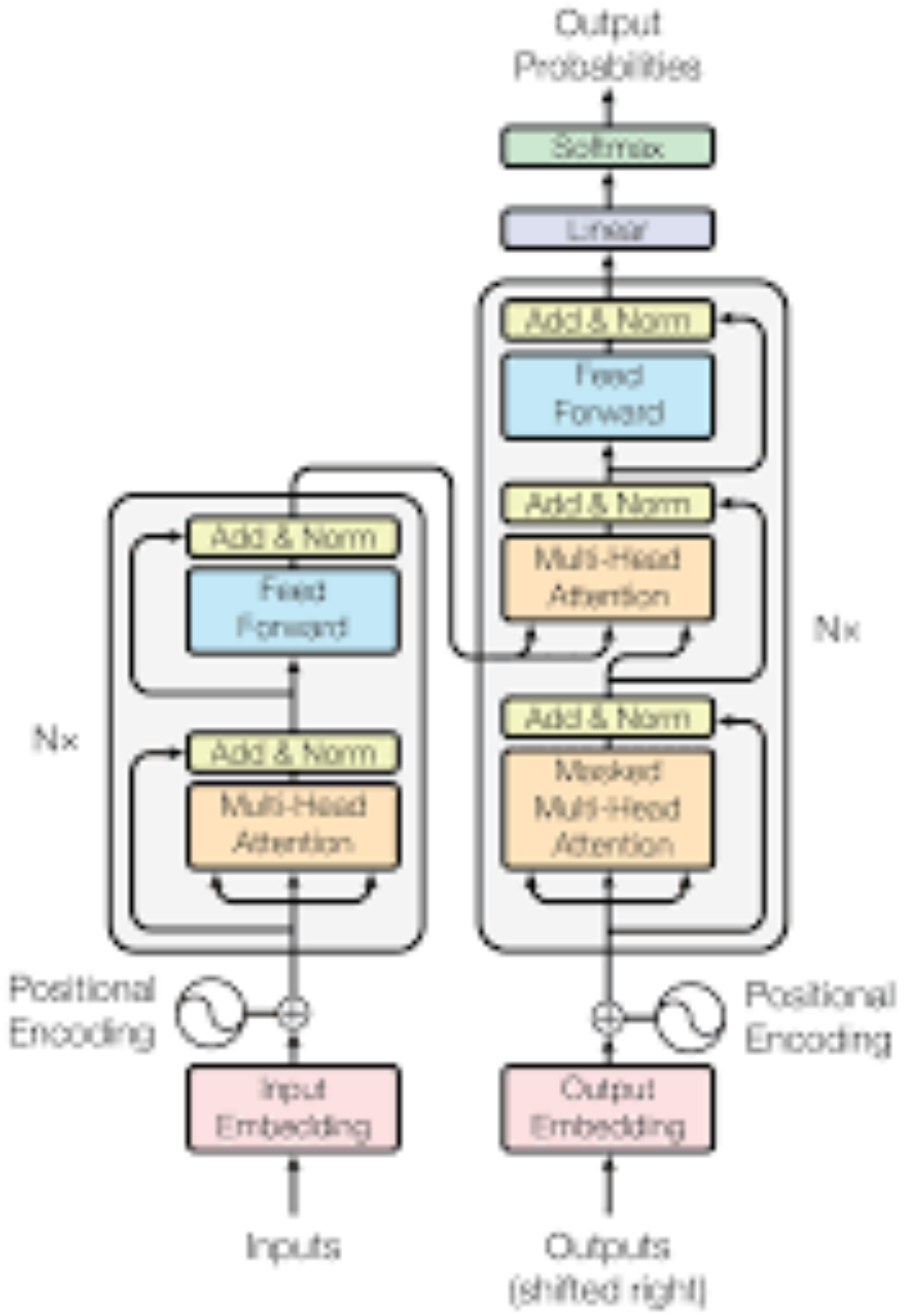
Difficult task for humans
- 50+ years of work

Sequence length
- difference in input/output

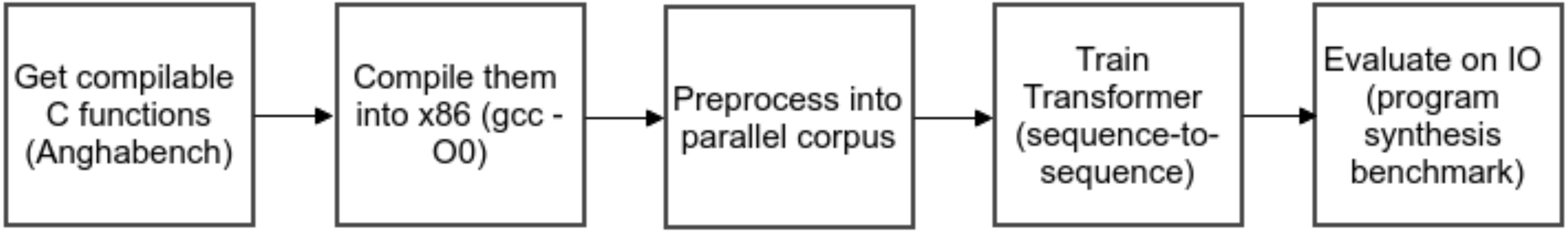
Training data
- where is it?

Evaluation





Neural Compilation: Train Seq2Seq C->x86



Results: Fib

Correct translation
- IO equivalent
- Does the same thing

Additional noop
-in ground truth
Additional moves
-in translation

C input

```
int fib_n(int n) {
  int i = 1;
  int r = 1;
  while (n > 1) {
    i = r - i;
    r = i + r;
    n = n - 1;
  }
  return r;
}
```

Model assembler

```
fib_n:
 .LFB0:
 .cfi_startproc
 pushq %rbp
 .cfi_def_cfa_offset 16
 .cfi_offset 6 , -16
 movq %rsp , %rbp
 .cfi_def_cfa_register 6
 movl %edi , -20 ( %rbp )
 movl $1 , -8 ( %rbp )
 movl $1 , -4 ( %rbp )
 jmp .L2
 .L3:
 movl -4 ( %rbp ), %eax
 subl -8 ( %rbp ), %eax
 movl %eax , -8 ( %rbp )
 movl -8 ( %rbp ), %edx
 movl -4 ( %rbp ), %eax
 addl %edx , %eax
 movl %eax , -4 ( %rbp )
 subl $1 , -20 ( %rbp )
 .L2:
 cmpl $1 , -20 ( %rbp )
 jg .L3
 movl -4 ( %rbp ), %eax
 popq %rbp
 .cfi_def_cfa 7 , 8
 ret
 .cfi_endproc
```

Ground truth assembler

```
fib_n:
.LFB0:
 .cfi_startproc
 endbr64
 pushq %rbp
 .cfi_def_cfa_offset 16
 .cfi_offset 6, -16
 movq %rsp, %rbp
 .cfi_def_cfa_register 6
 movl %edi, -20(%rbp)
 movl $1, -8(%rbp)
 movl $1, -4(%rbp)
 jmp  .L2
.L3:
 movl -4(%rbp), %eax
 subl -8(%rbp), %eax
 movl %eax, -8(%rbp)
 movl -8(%rbp), %eax
 addl %eax, -4(%rbp)
 subl $1, -20(%rbp)
.L2:
 cmpl $1, -20(%rbp)
 jg   .L3
 movl -4(%rbp), %eax
 popq %rbp
 .cfi_def_cfa 7, 8
 ret
 .cfi_endproc
```

Results: Search

Correct translation

Non- trivial
- Try doing this by hand!

Great that it can work
- but c.30% is far from practical
- so multi-modal training

C input

```
int search(int *arr,
           int v,
           int n) {
  int idx = -1;
  for (int i = 0; i < n;
       ++i) {
    if (idx == -1 &&
        arr[i] == v) {
      idx = i;
    }
  }
  return idx;
}
```

Model assembler

```
search:
 .LFB0:
 .cfi_startproc
 pushq %rbp
 .cfi_def_cfa_offset 16
 .cfi_offset 6, -16
 movq %rsp, %rbp
 .cfi_def_cfa_register 6
 movq %rdi, -24(%rbp)
 movl %esi, -28(%rbp)
 movl %edx , -32(%rbp)
 movl $-1, -8(%rbp)
 movl $0, -4(%rbp)
 jmp .L2
 .L4:
 cmpl $-1, -8(%rbp)
 jne .L3
 movl -4(%rbp), %eax
 movslq %eax, %rdx
 movq -24(%rbp), %rax
 addq %rdx, %rax
 movl (%rax), %eax
 cmpl -28(%rbp), %eax
 jne .L3
 movl -4(%rbp), %eax
 movl %eax, -8(%rbp)
 .L3:
 addl $1, -4(%rbp)
 .L2:
 movl -4(%rbp), %eax
 cmpl -32(%rbp), %eax
 jl .L4
 movl -8(%rbp), %eax
 popq %rbp
 .cfi_def_cfa 7, 8
 ret
 .cfi_endproc
```

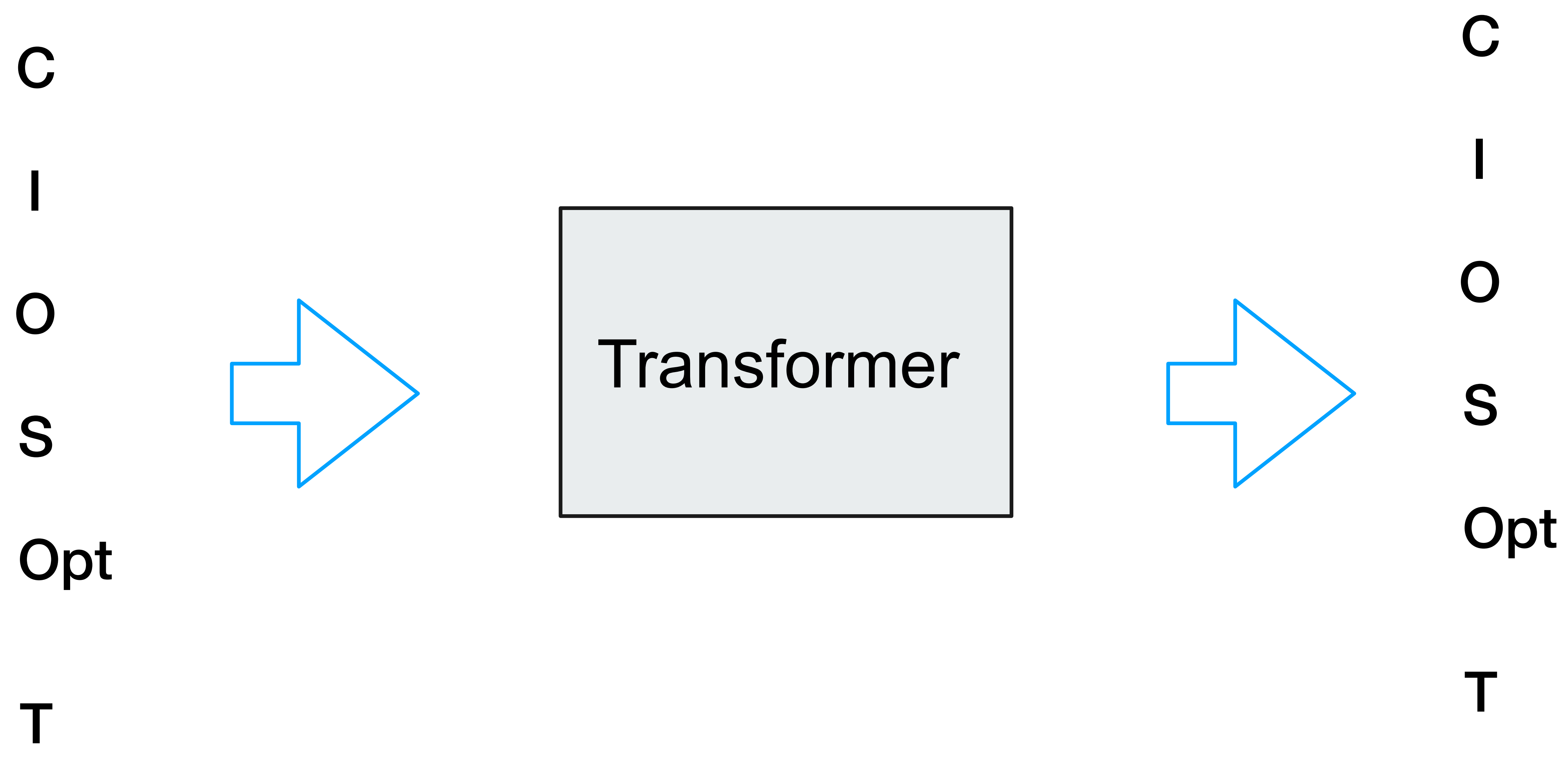
Ground truth assembler

```
search:
.LFB0:
 .cfi_startproc
 endbr64
 pushq %rbp
 .cfi_def_cfa_offset 16
 .cfi_offset 6, -16
 movq %rsp, %rbp
 .cfi_def_cfa_register 6
 movq %rdi, -24(%rbp)
 movl %esi, -28(%rbp)
 movl %edx, -32(%rbp)
 movl $-1, -8(%rbp)
 movl $0, -4(%rbp)
 jmp  .L2
.L4:
 cmpl $-1, -8(%rbp)
 jne  .L3
 movl -4(%rbp), %eax
 cltq
 leaq 0(,%rax,4), %rdx
 movq -24(%rbp), %rax
 addq %rdx, %rax
 movl (%rax), %eax
 cmpl %eax, -28(%rbp)
 jne  .L3
 movl -4(%rbp), %eax
 movl %eax, -8(%rbp)
.L3:
 addl $1, -4(%rbp)
.L2:
 movl -4(%rbp), %eax
 cmpl -32(%rbp), %eax
 jl   .L4
 movl -8(%rbp), %eax
 popq %rbp
 .cfi_def_cfa 7, 8
 ret
 .cfi_endproc
```

[AIPLANS@NeurIPS21]

Mult-lingual/modal translation

Build a multi-modal, multi-task model
Pose all tasks (including pre-training)
- with same format
- works multiple masks



C: C function, S,T: assemblers, I input example, O output example, Opt: optimise

<X> x1,x2.. xN </X> <Y> <mask> </Y> -> <mask> <END>

Types of translation

Compilation	C->s
Decompilation	s->C
Program Synthesis	I,O->C
Binary translation	arm <-> x86
Binary Optimisation	x86-> smaller x86
Evaluation	C,I->O
Latent evaluation	I,O,I->O

Zero -shot

- seen target but not direction in training

Zero++

- not seen targets
- eg arm -> smaller arm

Highly Preliminary Results

Compilation C->s:	56%
Decompilation s->C	27%
Program Synthesis I,O->C	21%
Evaluation C,I->O	47%
Latent evaluation I,O,I->O	39%

Currently
- training a larger model (1.5B+)

Using models to repair
- predict errors (lots of training data!)
- predict repair (using same/new model)

Uses ExeBench
- Expanded AnghaBench [MachineProgramming22@PLDI]

GitHub C code
- Executable code
- IO examples (autogen)

Highly Preliminary Results

Compilation C->s:	56%
Decompilation s->C	27%
Program Synthesis I,O->C	21%
Evaluation C,I->O	47%
Latent evaluation I,O,I->O	39%

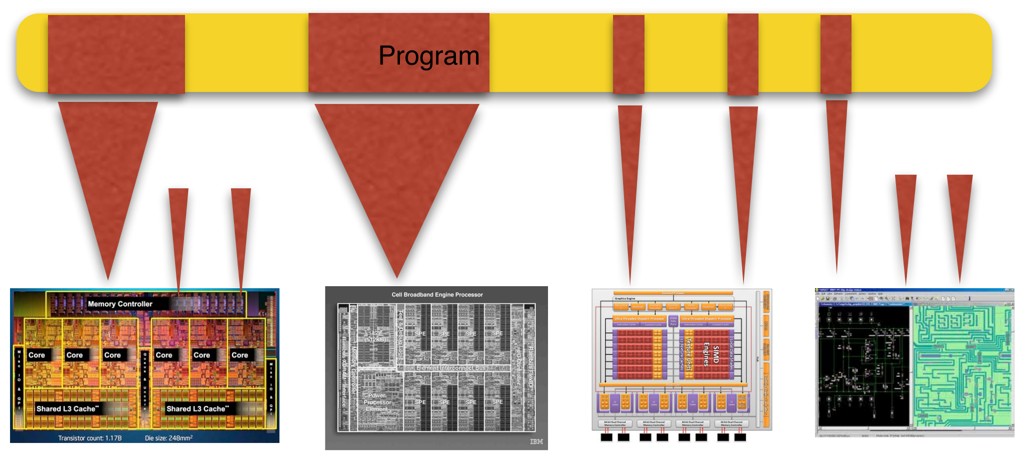
Currently
- training a larger model (1.5B+)

Using models to repair
- predict errors (lots of training data!)
- predict repair (using same/new model)

Uses ExeBench
- Expanded AnghaBench [MachineProgramming22@PLDI]

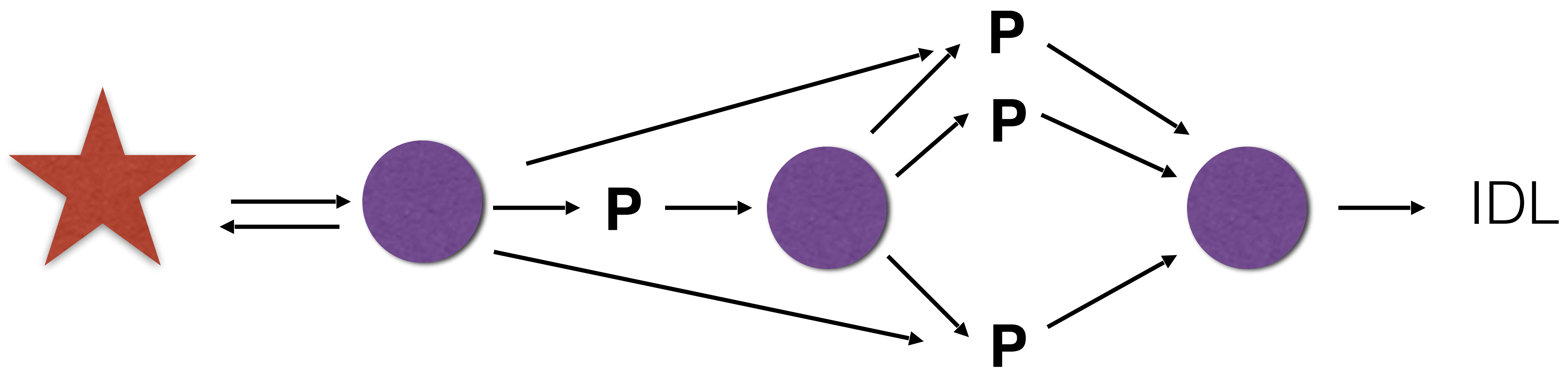
GitHub C code
- Executable code
- IO examples (autogen)

Conclusion

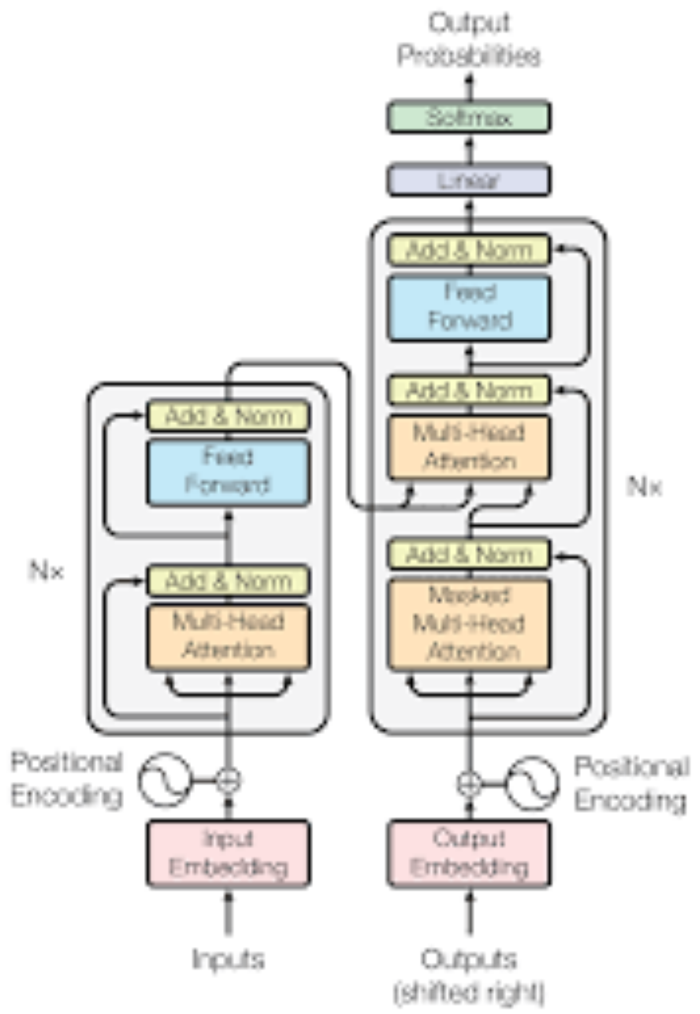


Matching Hardware to Software
- enables hardware innovation

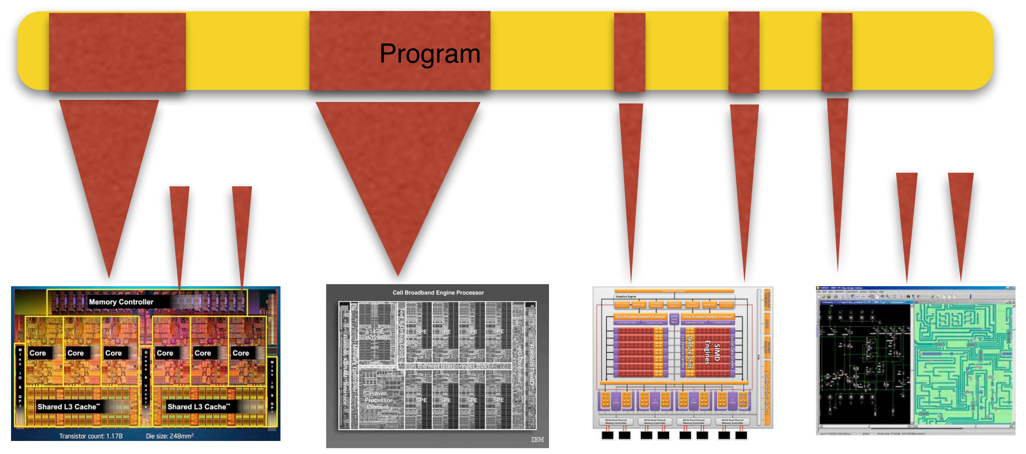
Program synthesis and code matching
- big step acceleration



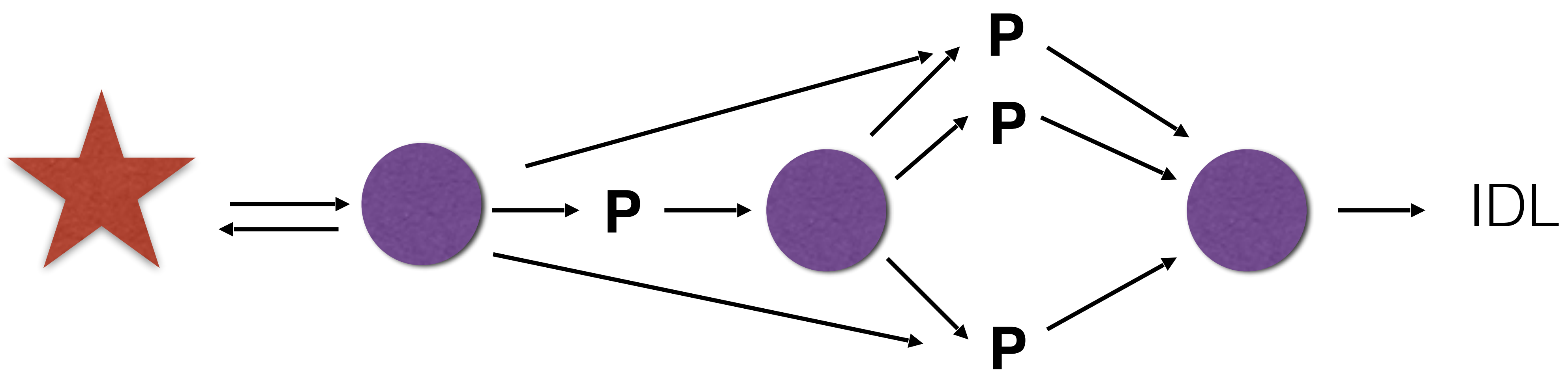
Going beyond simple acceleration requires new approaches
-compilation as neural machine translation



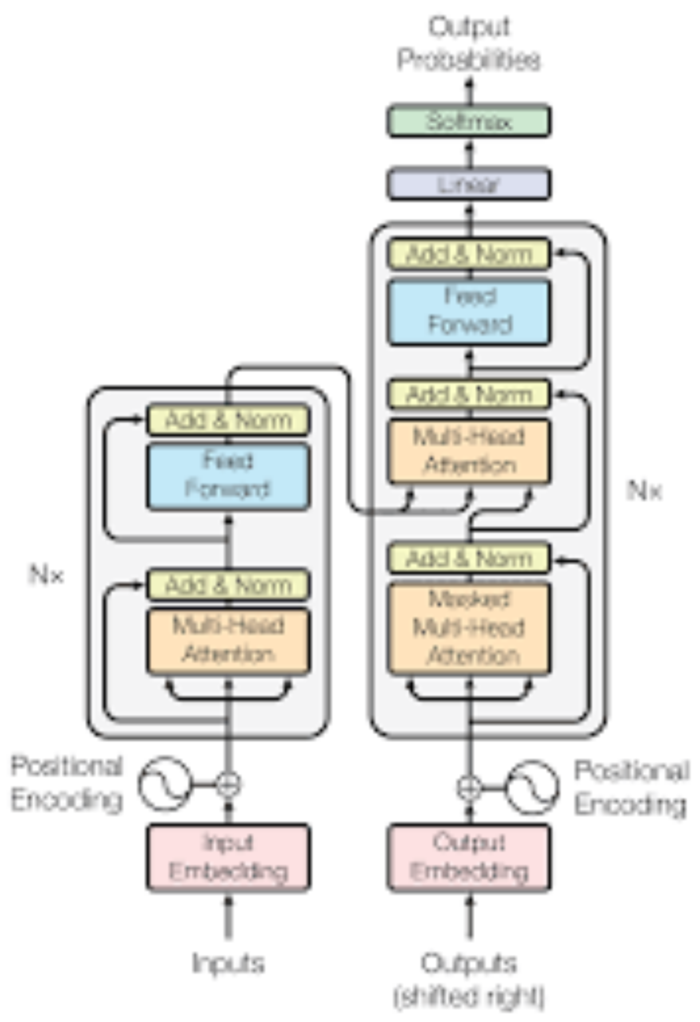
Conclusion



Matching Hardware to Software
- enables hardware innovation



Program synthesis and code matching
- big step acceleration



Going beyond simple acceleration requires new approaches
-compilation as neural machine translation

New technologies + endless automation = bridging software/hardware gap