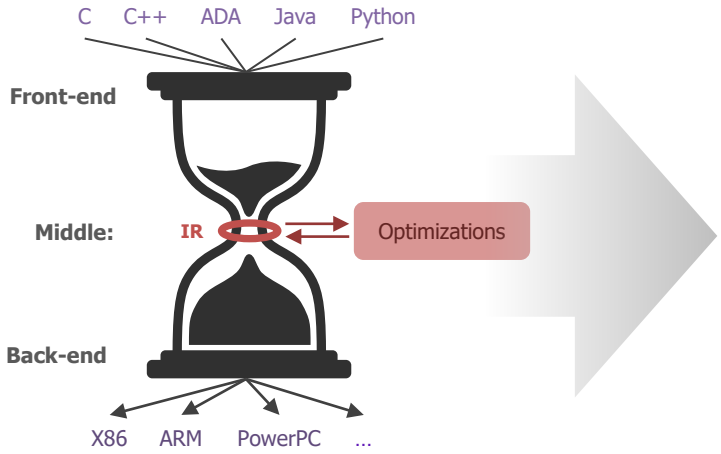
Machine learning and Optimization-driven Compilers for Heterogeneous Architectures: MOCHA

Howard Shrobe
DARPA
Information Innovation Office (I2O)

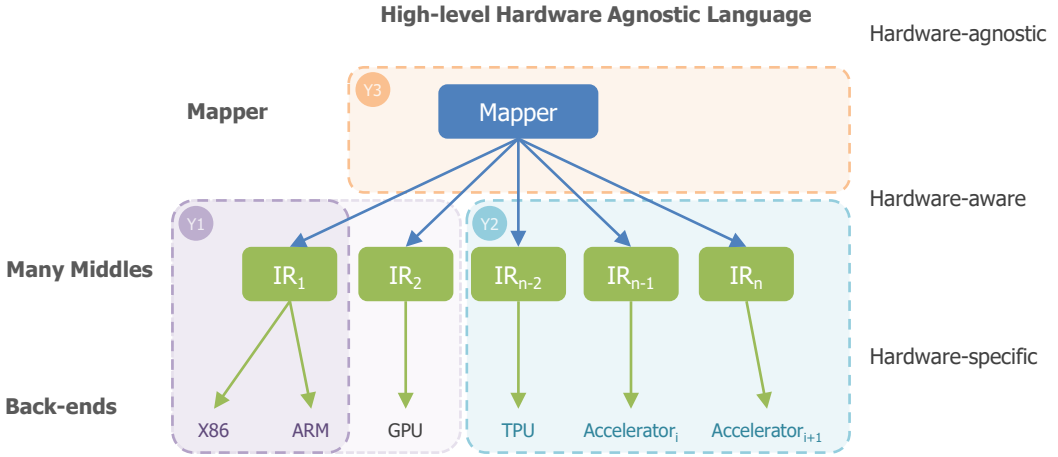
Machine learning and Optimization-guided Compilers for Heterogeneous Architectures (MOCHA)

Compilers can be rearchitected to generate highly efficient code for heterogeneous hardware platforms by employing ML-generated components that can be incorporated with minimal human effort

Today's compiler architecture



MOCHA



Apply ML across the entire compiler pipeline

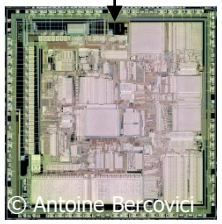
- Support multiple compute architectures simultaneously
- Enable holistic optimization across passes

IR: Intermediate Representation
CPU: Central Processing Unit
GPU: Graphical Processing Unit
TPU: Tensor Processing Unit

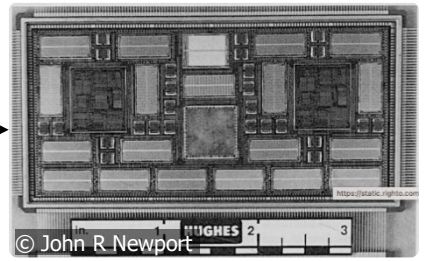
The tale of the F-22 "Software Crisis"

1985
F-22 development begins

1987
ADA Mandate



1988
Intel announces i960.
Pitched as ADA Processor



F-22 common integrated processor based on i960MX

1990
i960 development halted. Intel charged thousands of dollars for each i960 MX and each F-22 contained a cluster of 35 i960MX processors

Switch to PowerPC and then 386

Major software rewrites due to processor switch

1991

"The 1.55 million SLOC to be built for the F-22 represents the **most significant threat to the successful development of the aircraft**."

Defense Acquisition Board Report, June 1991

1997
F-22 first flight

2005
F-22 enters military service

March 25, 1998

"The Air Force is concerned that the writing and testing of avionics software may not be progressing sufficiently to provide software deliveries on schedule."

F-22 AIRCRAFT: Progress of the Engineering and Manufacturing Development Program
Statement of Louis J. Rodrigues, Director, Defense Acquisition Issues, National Security and International Affairs Division

"As of 2003, after over a decade of effort, the code was crashing, on average, every three hours. **Some parts of the software system were failing every 90 minutes.**"

https://freerepublic.com/focus/f-news/1110502/posts

SLOC: Source Lines Of Code

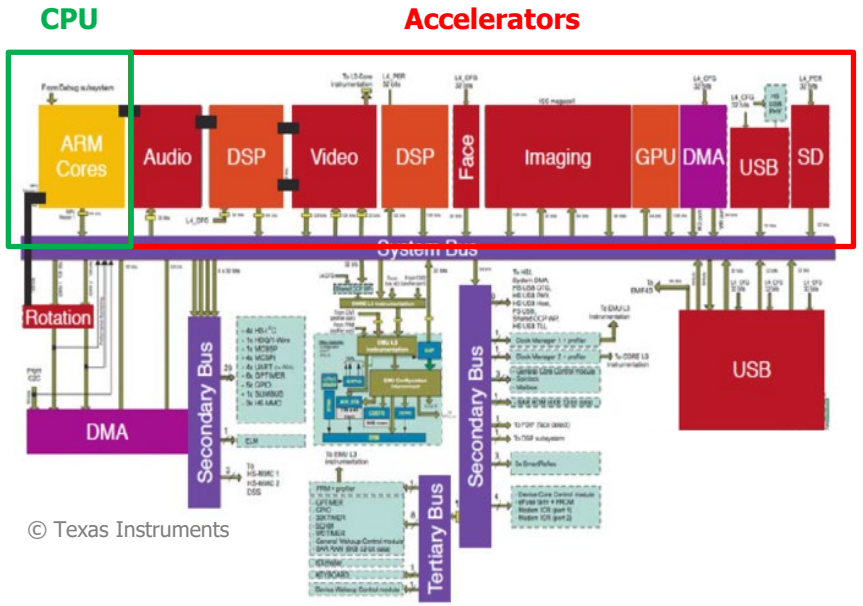
The lesson of F-22 software

- DoD system design, development and deployments happen over very long timeframes
- Processors come and go much more quickly
  - Even new versions of existing components may have wildly different micro-architecture with different performance profiles
- Mission needs change much more quickly
- Writing software "close to the metal" to gain performance impedes agility and adaptability
- Mission software should be **"hardware agnostic"** and written at a high level of abstraction
- The **compiler should decide** how to utilize different processors and accelerators, enabling the software to be remapped to take advantage of new hardware
  - Increases agility
  - Requires much less (scarce) human resources
  - Increases software reliability and correctness

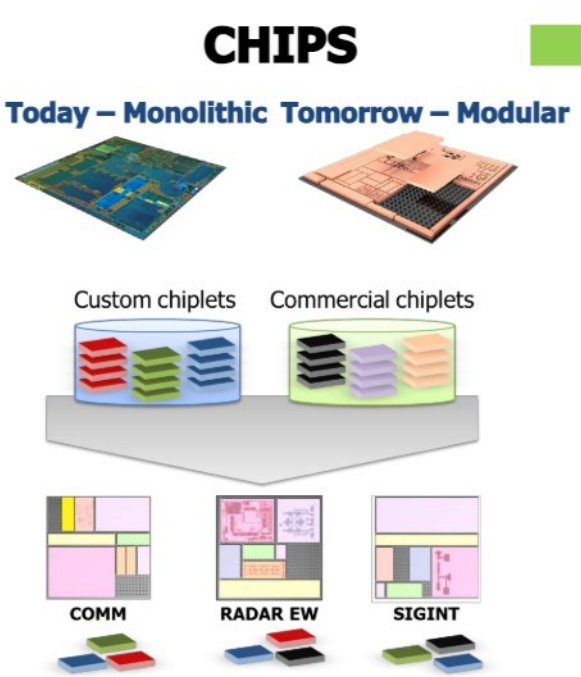
Today: State-of-the-art architectures include heterogeneity

End of Moore's Law has led to proliferation of processing units and created a need for specialized computer architectures and accelerators

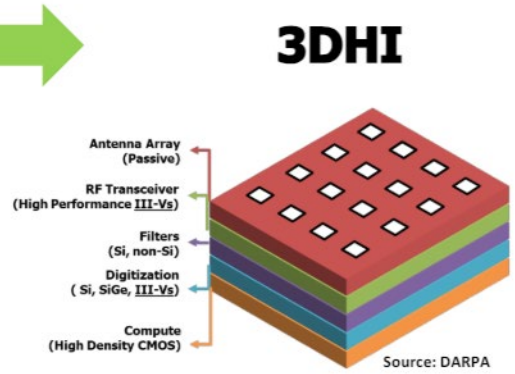
2015



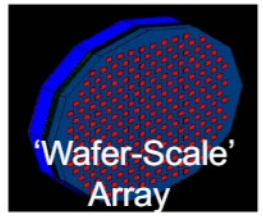
2024 and beyond



Source: DARPA CHIPS BAA



Source: DARPA



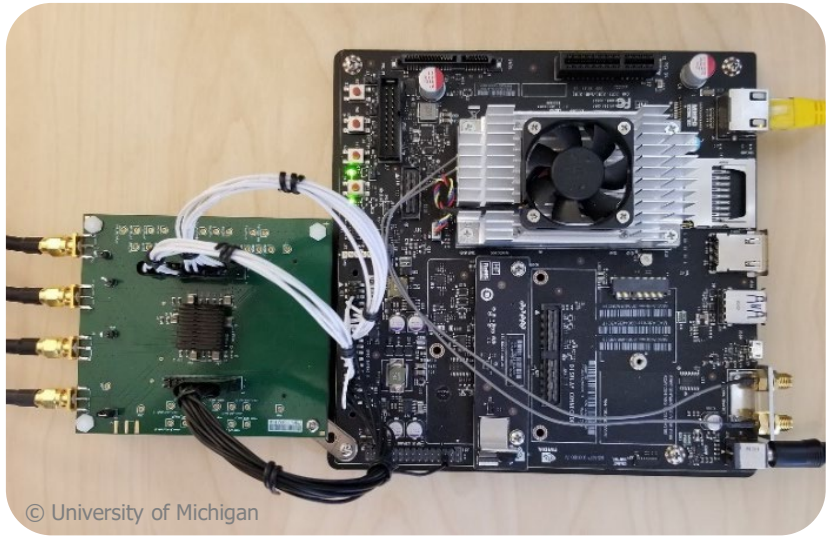
Source: Kazior: IEEE 2016 International Microwave Symposium Workshop MG: Heterogeneous integration of Silicon RFIC with III-V

Open Multimedia Applications Platform system on chip: TI OMAP4 SoC

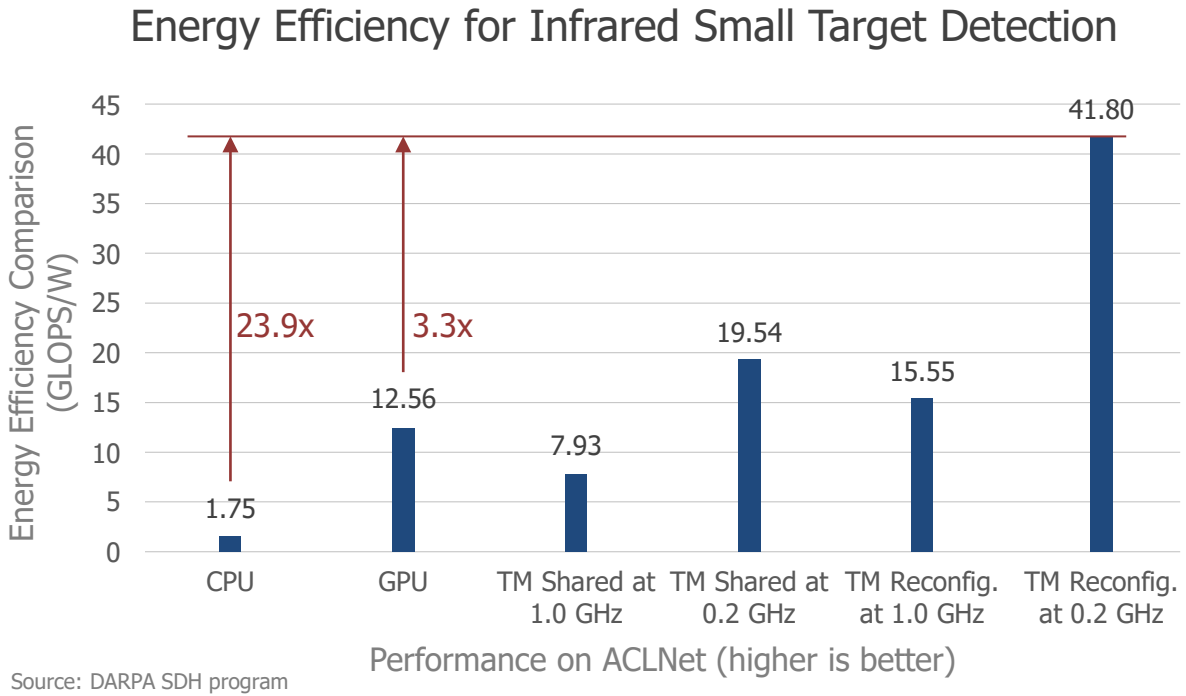
- More computational power on accelerators than on CPU cores
- Specialization overcomes power limitations
- Programming heterogenous systems is difficult

Today: Heterogeneous hardware overcomes challenging SWaP constraints

With Compiler Support, Heterogeneous and Specialized SoC Outperforms CPU's and GPU's



Transmuter v.5 SoC
(DARPA MTO SDH Program UMich Team)



Source: DARPA SDH program

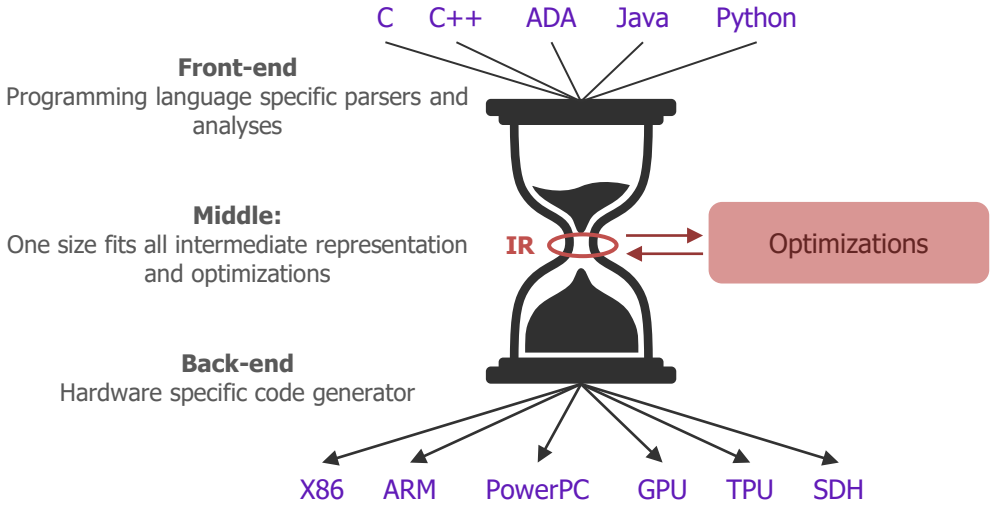
Manual development of compiler tool chain required

SoC: System on Chip
SDH: Software-Defined Hardware
SWaP: Size, Weight, and Power

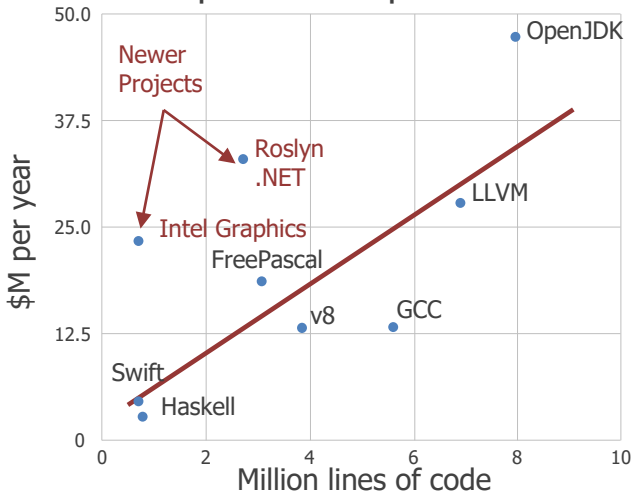
Pal, Subhankar, et al. "Transmuter: Bridging the efficiency gap using memory and dataflow reconfiguration." *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. 2020.

Today: Compiler development is dependent on manual effort

The Classic "Hourglass" Architecture



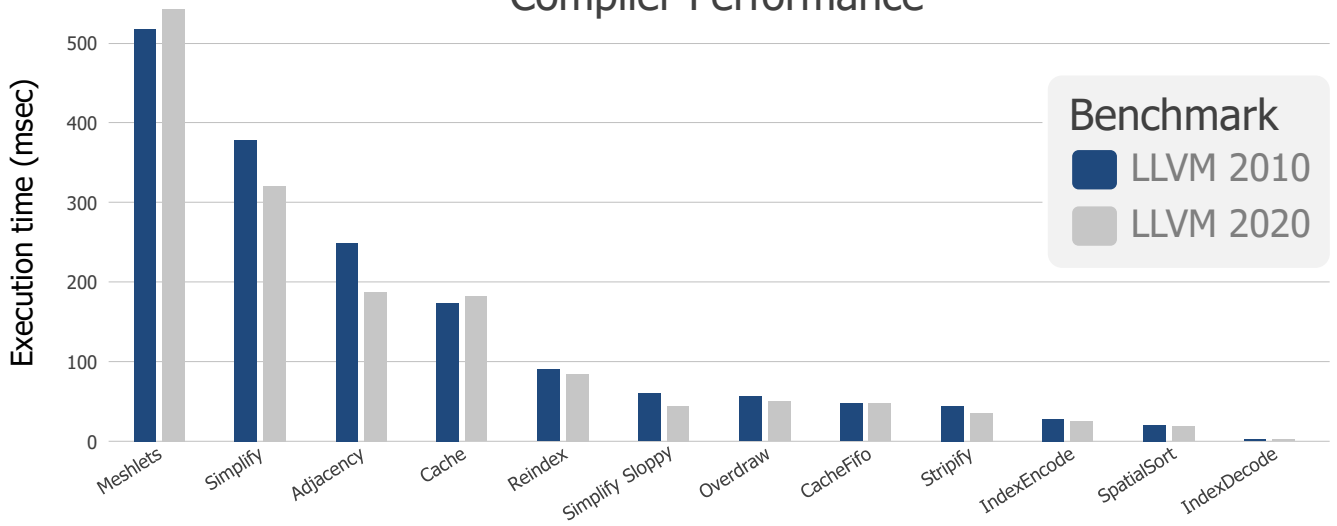
Compiler development cost



Source: Chris Cummins, Deep Learning for Compilers 2019

- Unresponsive to hardware innovations
- Abstraction of the standard CPU architecture
- Based on classic Intermediate Representations (IR), analyses, and transformations; ad hoc heuristics; and hand-tuned optimizations
- Manufactures spec-sheets are inaccurate due to micro-architectural interactions
- Don't match the capabilities of accelerators
- Optimization design is a "black art"

Compiler Performance



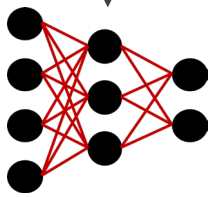
Source: Arseny Kapoulkine https://zeux.io/2022/01/08/on-proebstings-law/

Challenge: Compiling for heterogeneous ensembles with limited human effort

Flight Controls

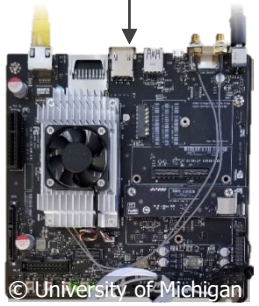
CPU

Signal Processing



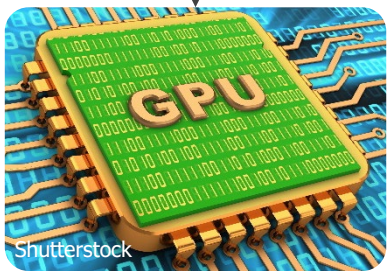
DSP

Object Detection



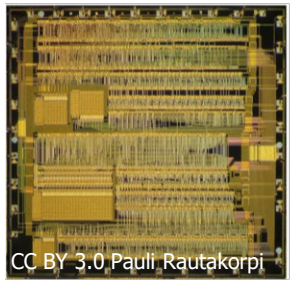
Transmuter

Graphics



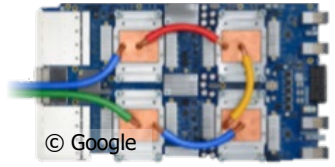
GPU

Security



Crypto Processor

Object Classification

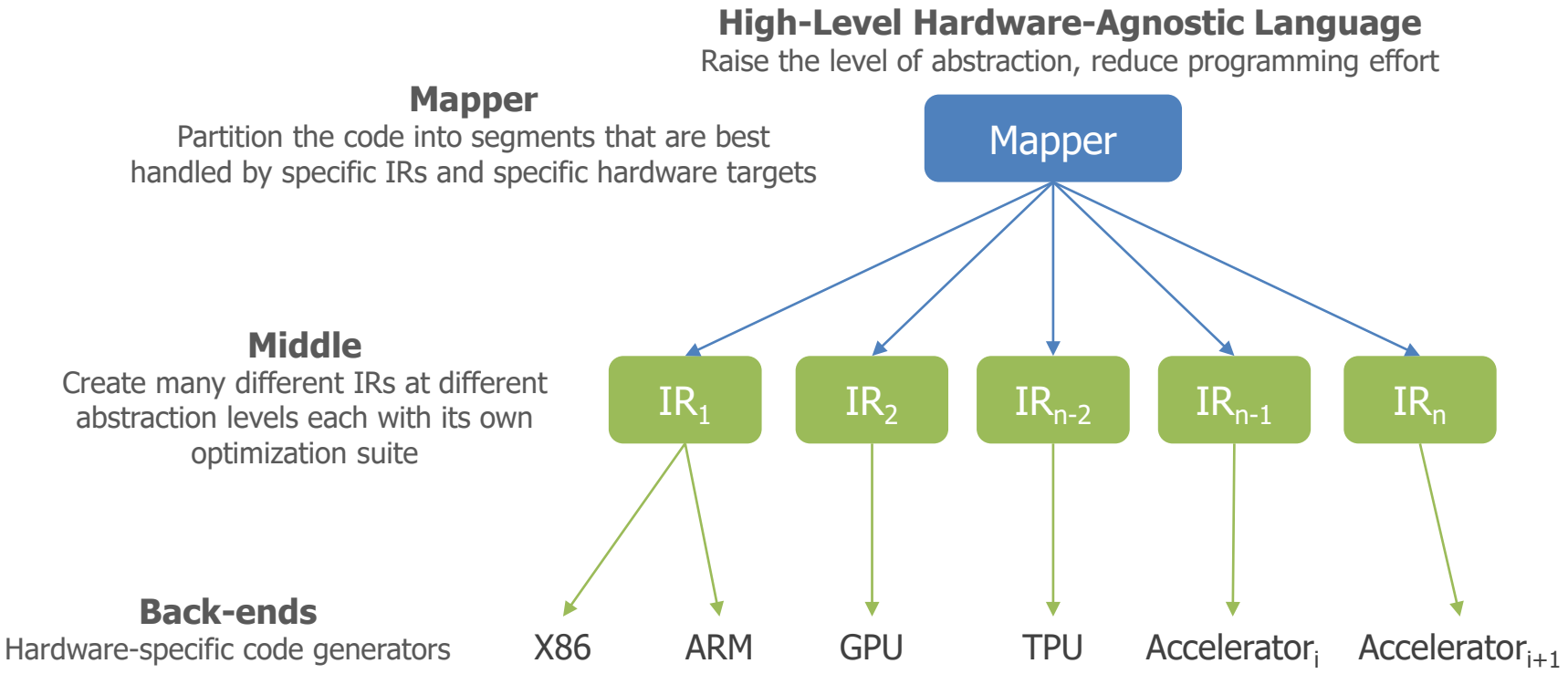


TPU

CPU: Central Processing Unit
DSP: Digital Signal Processor
GPU: Graphical Processing Unit
TPU: Tensor Processing Unit

Tomorrow: Machine learning can break the manual bottlenecks

Technical hypothesis: Compilers can be rearchitected to generate highly efficient code for heterogeneous hardware platforms by using ML-generated components that can be incorporated with minimal human effort



Replace the "Hourglass"

Apply Machine Learning Throughout

CPU: Central Processing Unit
GPU: Graphics Processing Unit
TPU: Tensor Processing Unit
IR: Intermediate Representation

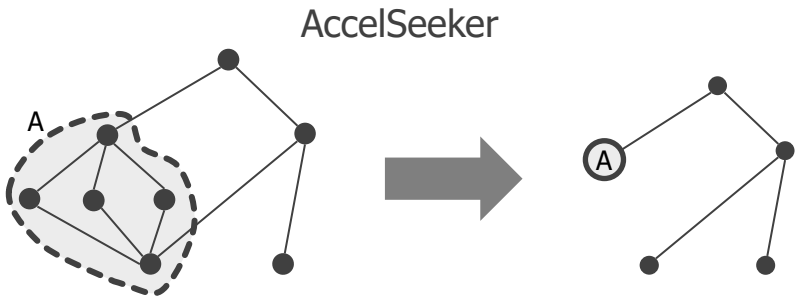
Approach:

- Move to "Hardware Agnostic" programming
- Reduce human effort in compiler construction by using Machine Learning to generate compiler components
- Target a variety of accelerators, producing correct, highly optimized code for a heterogenous ensemble of components

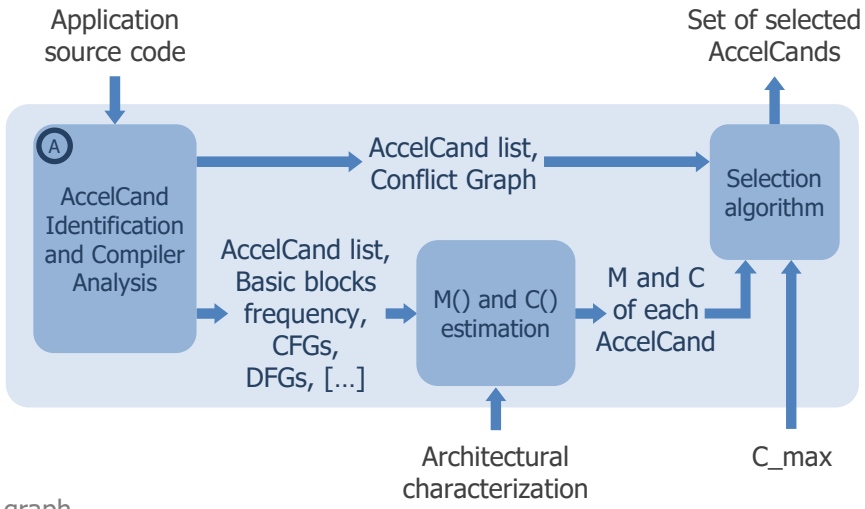
Reason for confidence:
AccelSeeker: Identifying acceleration candidates

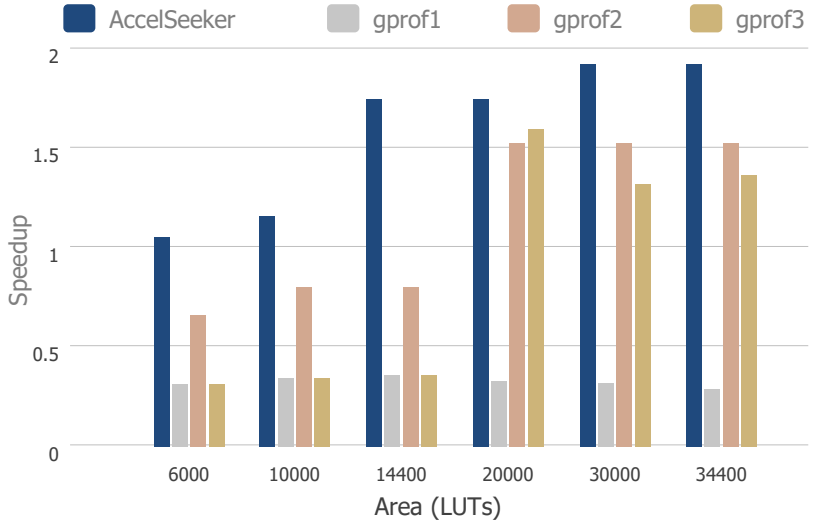
Front-end

- Front-end components need to identify regions of the code that can be targeted to accelerators
- Candidate regions identified by examining the call graph, which is constructed in the compiler front end
- Performance models identify which sub-graphs are candidates for acceleration



Candidate is replaced in call graph





Speedup of H.264 decoder by implementing identified hardware accelerators

Limitations:
Performance models are ad hoc and not general

Zacharopoulos, Georgios, et al. "Compiler-assisted selection of hardware acceleration candidates from application source code." *2019 IEEE 37th International Conference on Computer Design (ICCD)*. IEEE, 2019.

CFG: Control-flow graph
DFG: Data-flow graph
LUT: Logical Unit

Reason for confidence:
VeGen: Learning what optimization to apply

Middle

```
132604 <intrinsic tech="MMX" name="_m_pmaddwd">
132605     <type>Integer</type>
132606     <CPUID>MMX</CPUID>
132607     <category>Arithmetic</category>
132608     <return type="__m64" varname="dst" etype="FP32"/>
132609     <parameter type="__m64" varname="a" etype="SI64"/>
132610     <parameter type="__m64" varname="b" etype="SI64"/>
132611     <description>Multiply packed signed 16-bit integers in "a"
           and "b", producing intermediate signed 32-bit integers.
           Horizontally add adjacent pairs of intermediate 32-bit integers,
           and pack the results in "dst".</description>
132612     <operation>
132613 FOR j := 0 to 1
132614     i := j*32
132615     dst[i+31:i] := SignExtend32(a[i+31:i+16]*b[i+31:i+16]) +
           SignExtend32(a[i+15:i]*b[i+15:i])
132616 ENDFOR
132617     </operation>
132618     <instruction name="PMADDWD" form="mm, mm"
           xed="PMADDWD_MMXq_MMXq"/>
132619     <header>mmintrin.h</header>
132620 </intrinsic>
```

Architecture Manual

Scalar Program

```
int16_t A[4], B[4];
int32_t C[2];
void dot_prod() {
    C[0] = A[0] * B[0] + A[1] * B[1];
    C[1] = A[2] * B[2] + A[3] * B[3];
}
```

Target-specific Vectorizer

```
vmovd  xmm0, [A]
vmovd  xmm1, [B]
pmaddwd xmm0, xmm1, xmm0
vmovd  [C], xmm0
```

Vector Program

VeGen = Vectorizer Generator

Chen, Yishen, et al. "VeGen: a vectorizer generator for SIMD and beyond." *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2021.

VeGen: ML generated code is more compact and efficient

A matrix-vector multiplication kernel from TVM

```
void
dot_16x1x16_uint8_int8_int32(
 uint8_t data[restrict 4],
 int8_t kernel[restrict 16][4],
 int32_t output[restrict 16]) {
  for (int i = 0; i < 16; i++)
    for (int k = 0; k < 4; k++)
      output[i] +=
        data[k] * kernel[i][k];
}
```

ICC (1x)

```
movzx     r11d, [rdi]
movsx     eax, [rsi]
imul      r11d, eax
movsx     ecx, [1+rsi]
movzx     r8d, [1+rdi]
imul      r8d, ecx
movzx     r10d, [2+rdi]
movsx     r9d, [2+rsi]
imul      r10d, r9d
movsx     eax, [3+rsi]
movzx     ecx, [3+rdi]
imul      ecx, eax
add       r11d, [rdx]
add       r11d, r8d
add       r11d, r10d
add       r11d, ecx
mov       [rdx], r11d
movzx     r9d, [rdi]
movsx     r11d, [4+rsi]
imul      r9d, r11d
movsx     eax, [5+rsi]
movzx     r8d, [1+rdi]
imul      r8d, eax
movzx     eax, [2+rdi]
movsx     ecx, [6+rsi]
imul      eax, ecx
movsx     r10d, [7+rsi]
movzx     r11d, [3+rdi]
imul      r11d, r10d
add       r9d, [4+rdx]
add       r9d, r8d
add       r9d, eax
add       r9d, r11d
mov       [4+rdx], r9d
movzx     r8d, [rdi]
movsx     r9d, [8+rsi]
imul      r8d, r9d
movsx     eax, [9+rsi]
```

GCC (1.5x)

```
vmovdqa xmm0, .LC0[rip]
vmovdqu xmm1, [rsi]
vmovdqu xmm3, [rsi+16]
vmovdqu xmm2, [rsi+32]
vmovdqu xmm6, [rsi+48]
vpand xmm10, xmm0, xmm1
vpsrlw xmm1, xmm1, 8
movzx r9d, [rdi]
vpand xmm4, xmm0, xmm3
vpsrlw xmm3, xmm3, 8
movzx ecx, [rdi+2]
movzx eax, [rdi+3]
vpackuswb xmm4, xmm10, xmm4
vpackuswb xmm1, xmm1, xmm3
vpand xmm10, xmm0, xmm2
movzx r8d, [rdi+1]
vpand xmm3, xmm0, xmm6
vpsrlw xmm2, xmm2, 8
vmovd xmm8, r9d
vpackuswb xmm3, xmm10, xmm3
vpsrlw xmm6, xmm6, 8
vpand xmm10, xmm0, xmm4
vpackuswb xmm2, xmm2, xmm6
vpsrlw xmm4, xmm4, 8
vpand xmm6, xmm0, xmm3
vpsrlw xmm3, xmm3, 8
vpackuswb xmm10, xmm10, xmm6
vpbroadcastw xmm8, xmm8
vpackuswb xmm6, xmm4, xmm3
vpand xmm3, xmm0, xmm1
vpand xmm0, xmm0, xmm2
vpackuswb xmm3, xmm3, xmm0
vpsrldq xmm0, xmm10, 8
vpmovsxbw xmm4, xmm10
vmovd xmm7, r8d
vpmullw xmm4, xmm4, xmm8
vpmovsxbw xmm0, xmm0
vpmullw xmm0, xmm0, xmm8
```

Clang/LLVM (2.2x)

```
movzx eax, [rdi + 3]
vpbroadcastd zmm8, eax
movzx eax, [rdi + 2]
vpbroadcastd zmm9, eax
movzx eax, [rdi + 1]
vpbroadcastd zmm10, eax
movzx eax, [rdi]
vpbroadcastd zmm11, eax
vmovdqu xmm0, [rsi]
vmovdqu xmm1, [rsi + 16]
vmovdqu xmm6, [rsi + 32]
vmovdqu xmm7, [rsi + 48]
vmovdqa xmm2, [rip + .LCPI0_0]
vpshufb xmm3, xmm7, xmm2
vpshufb xmm2, xmm6, xmm2
vpunpckldq xmm2, xmm2, xmm3
vmovdqa xmm3, [rip + .LCPI0_1]
vpshufb xmm4, xmm1, xmm3
vpshufb xmm3, xmm0, xmm3
vpunpckldq xmm3, xmm3, xmm4
vpblendd xmm12, xmm3, xmm2, 12
vmovdqa xmm3, [rip + .LCPI0_2]
vpshufb xmm4, xmm7, xmm3
vpshufb xmm3, xmm6, xmm3
vpunpckldq xmm3, xmm3, xmm4
vmovdqa xmm4, [rip + .LCPI0_3]
vpshufb xmm5, xmm1, xmm4
vpshufb xmm4, xmm0, xmm4
vpunpckldq xmm4, xmm4, xmm5
vpblendd xmm3, xmm4, xmm3, 12
vmovdqa xmm4, [rip + .LCPI0_4]
vpshufb xmm5, xmm7, xmm4
vpshufb xmm4, xmm6, xmm4
vpunpckldq xmm4, xmm4, xmm5
vmovdqa xmm5, [rip + .LCPI0_5]
vpshufb xmm2, xmm1, xmm5
vpshufb xmm5, xmm0, xmm5
vpunpckldq xmm2, xmm5, xmm2
vpblendd xmm2, xmm4, xmm2, 12
vmovdqa xmm4, [rip + .LCPI0_6]
vpshufb xmm5, xmm7, xmm4
```

VeGen (11x)

```
vmovdqu64     zmm0, [rdx]
vpbroadcastd  zmm1, [rdi]
vpdpbusd      zmm0, zmm0, [rsi]
vmovdqu64     [rdx], zmm0
```

Limitations:
Only one type of optimization

ICC: Intel C Compiler
GCC: Gnu C Compiler
Clang: LLVM compiler front-end
TVM: Tensor Virtual Machine
VeGen: Vectorizer Generator

Reason for confidence:
Ithemal: Automatic construction of accurate processor performance models

Back-end

Training Time

Execution Time

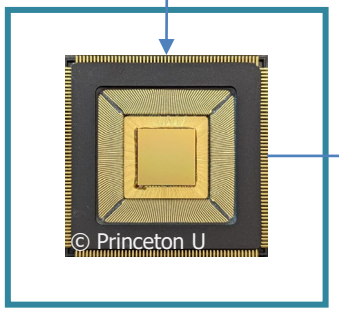
Code Generator

Code from Middle End

```
lea     r14, [rbx-0x40]
        ........
        ........
lea     rdx, [rbp+0x38]
cmp     rdi, rax
```

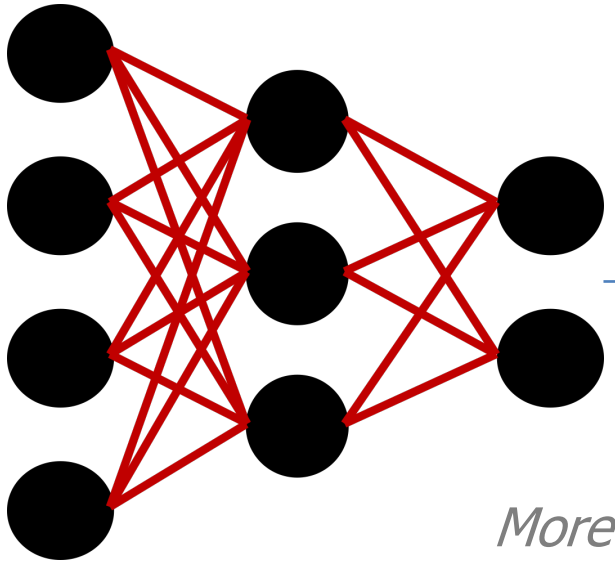
Code Sequences

Labeled Data Set



Measurements

ML Model



Trained Model

Model Prediction
≈ 44 cycles

Faster and easier to produce than models developed by hand.

More accurate performance model and therefore better code.

Ithemal (Instruction THroughput Estimator using MAchine Learning)
Mendis et al. [ICML'19]

Ithemal: ML used in back-end development improves code performance

Accuracy of Performance Model

Architecture	Predictor	Error
Ivy Bridge	Default	33.5%
	DiffTune	25.4% ± 0.5%
	Ithemal	9.4%
	IACA	15.7%
	OpenTuner	102.0%
Haswell	Default	25.0%
	DiffTune	23.7% ± 1.5%
	Ithemal	9.2%
	IACA	17.1%
	OpenTuner	105.4%
Skylake	Default	26.7%
	DiffTune	23.0% ± 1.4%
	Ithemal	9.3%
	IACA	14.3%
	OpenTuner	113.0%

Smaller is better

Throughput of Compiled Code

Method	Throughput (Instructions / second)
llvm-mca	492
IACA	541
Ithemal	560
Empirical execution	13

Bigger is better

Limitations:
- Only one architecture
- Only one figure of merit

Renda, A., Chen, Y., Mendis, C., & Carbin, M. (2020, October). Difftune: Optimizing cpu simulator parameters with learned differentiable surrogates. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (pp. 442-455). IEEE.

Program Structure

1. **ML-based Generators**: Multiple teams produce ML-based generators for different parts of compiler framework and one or two teams create a full ML-optimized tool chain
    1. Mapping
    2. Choosing and sequencing optimizations
    3. Performance Modeling appropriate for front end, middle, and back end
    4. Validation
    5. Code generation

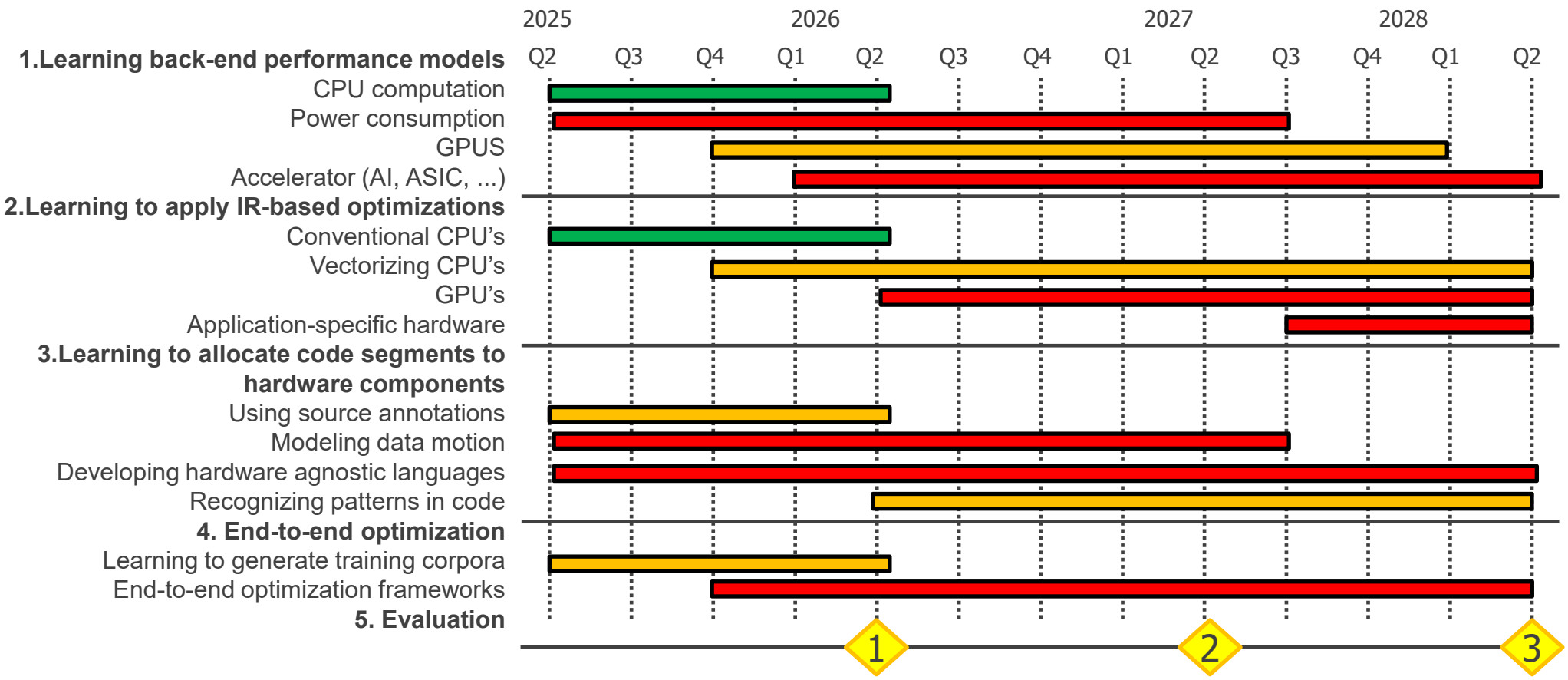
    Parts

    6. Integrated solutions: end-to-end compiler toolchains generated by the performer's and others' generators — Full

2. **Evaluation:** Government team creates a series of challenge problems of increasing complexity and measures component and system performance
    1. Using an ensemble of available hardware components and conventional languages
    2. Using models of emerging platforms
    3. Using hardware-agnostic languages

3. **Consortium Development:** creates an institutional framework to incentivize industry adoption and sustainment of the technology

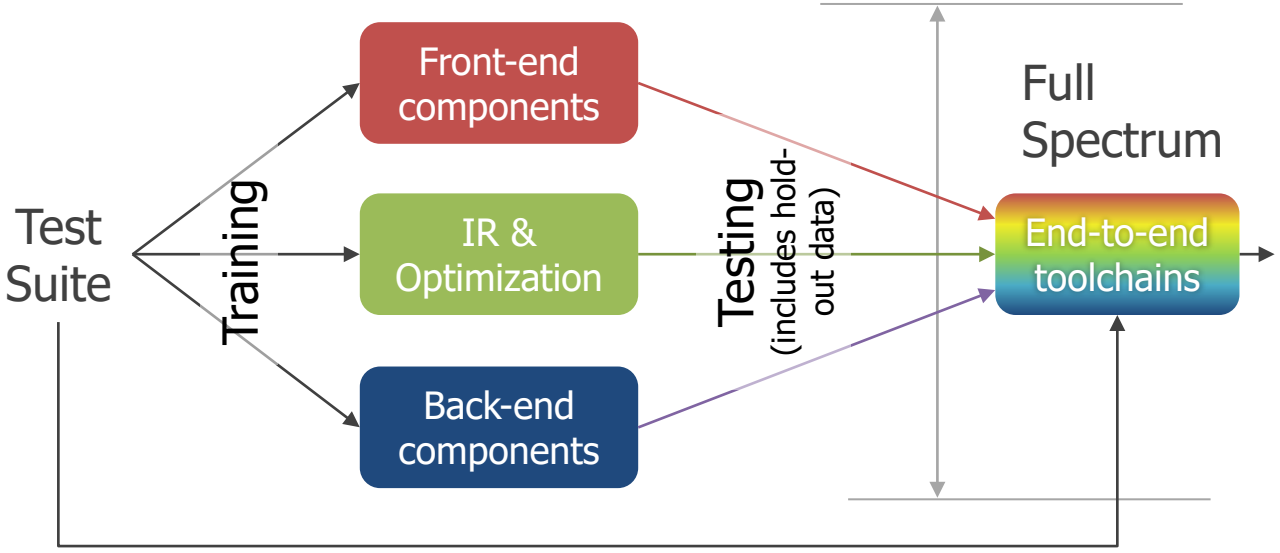
Program Timeline



- "Learning back-end performance models" (Back-End) doesn't rely on results in other areas and will provide capabilities for current systems (CPU's GPU's) in year 1 and will extend to other types of components in years 2 & 3
- "Learning to apply IR-based optimizations" (Middle) relies on back-end performance models and will provide capabilities in year 2 with extensions in year 3
- "Learning to allocate code segments to hardware components" (Front-End) depends on performance models from Middle. Final results in Year 3
- "End-to-end optimization" depends on all other technologies. Experimentation will proceed throughout with synthetic data. Final results in Year 3

Experimental design

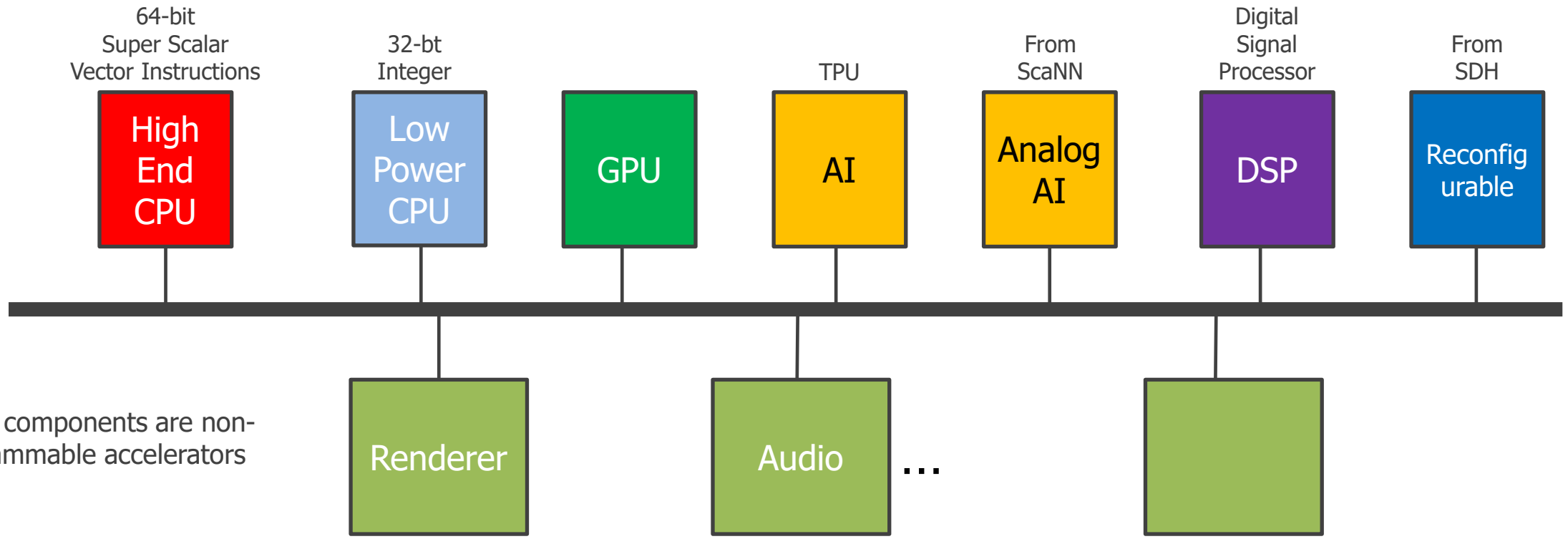
1. Government evaluation support team creates a **"test suite"** of system software and test criteria (e.g., speed, power, code size)
2. Technical teams generate components for **Front-end**, **Middle**, and **Back-end** and integrate these into end-to-end solutions
3. Evaluation support team runs the **full spectrum systems** and **measures performance** and lines of compiler instructions needed to configure the full spectrum systems.



Measurements:
- Human-written lines of compiler code needed
- Performance on test criteria

IR: Intermediate Representation

Notional Final Challenge Configuration



- The Year 3 challenge platforms will include an ensemble including 6 or more types of computational elements (and potentially more than 1 from each category)
- A variety of different configurations will be used as challenge platforms
- A variety of different programs will be compiled for these platforms

Experimental design – Evolution of tests

3 cycles of evaluation

- A mini test at 6-month boundaries (1.1, 2.1, 3.1)
- A full test at yearly boundaries (1.2, 2.2, 3.2)

Year 1 focuses on increasing the number of available accelerators

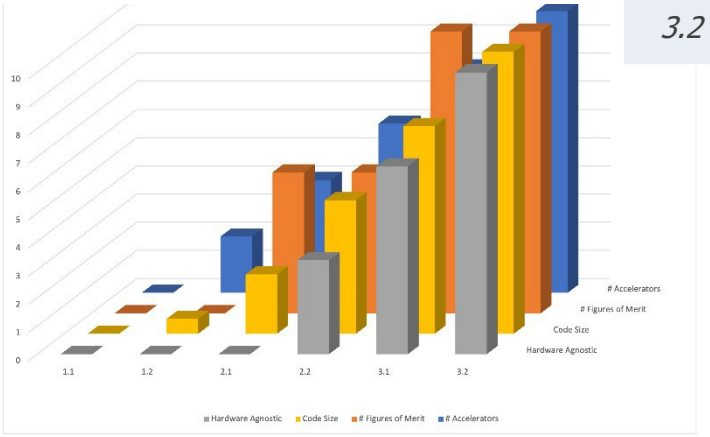
Year 2 add increasing complexity of the figure of merit (i.e., # of criteria – speed, power, code size)

Year 3 emphasizes the degree of hardware independence and application code size

Eval #	*# Component Types*	*Total # Components*	*Criteria in Figure of Merit*	*Hardware Agnostic*	*Source Code Size*
1.1	*2*	*2*	*1*	*75%*	*10,000*
1.2	*2*	*4*	*1*	*75%*	*25,000*
2.1	*3*	*6*	*2*	*75%*	*50,000*
2.2	*4*	*8*	*2*	*80%*	*100,000*
3.1	*5*	*10*	*3*	*85%*	*500,000*
3.2	*6*	*12*	*3*	*90%*	*1,000,000*

MOCHA: ML and Optimization guided compilers that require minimal human effort to generate efficient code for heterogeneous hardware platforms

Experimental Conditions as % of Final Goal Conditions



Year 3 Challenge problems include:
- Hardware agnostic languages
- All of the below

Year 2 Challenge problems include:
- Simulation models (or real systems) of emerging components in addition to those below

Year 1 Challenge problems include:
- Conventional programming languages
- Heterogeneous ensemble of existing hardware targets (CPUs, GPUs, …)

Metrics

1. Human effort reduction:
   - Time to create new compiler components
   - Measured by reduction in hand-written lines of compiler instructions

2. Code Quality multi-dimensional performance gain on benchmarks vs. baseline without ML guidance
   - Execution speed
   - Power consumption
   - Memory footprint

Metric	Year 1	Year 2	Year 3
Human Effort Reduction	50%	75%	90%
Code quality performance gain	1x	2.5x	5x

www.darpa.mil