VLSI architecture, synthesis & technology

UCLA

Democratizing Customized Computing

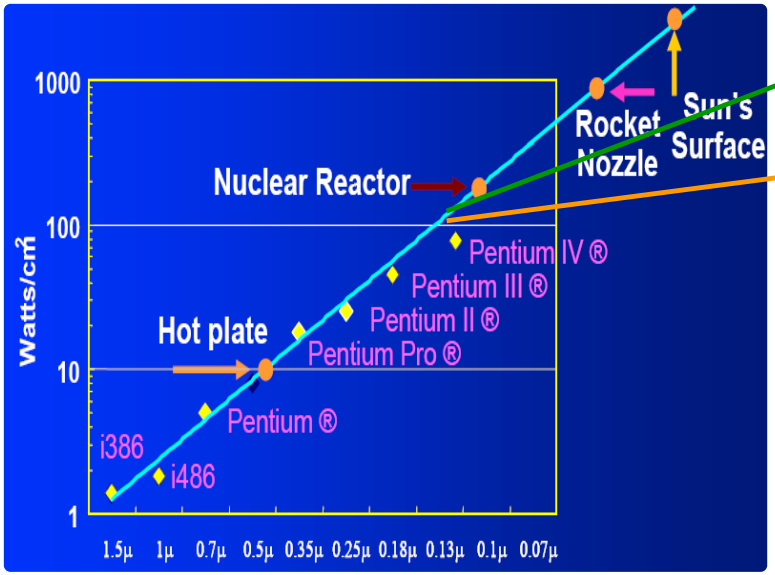
Jason Cong

Volgenau Chair for Engineering Excellence, UCLA Computer Science

Director, Center for Domain-Specific Computing (CDSC)

https://vast.cs.ucla.edu/people/faculty/jason-cong

From Parallization to Customization



Original source: Shekhar Borkar, Intel

Parallelization

Customization

Adapt the architecture to Application domain

2009 NSF Expeditions in Computing Award

UCLA Newsroom

UCLA Newsroom > All stories > News Releases

NSF awards UCLA $10 million to create customized computing technology

By Wileen Wong Kromhout | 8/11/2009 9:45:00 AM

The UCLA Henry Samueli School of Engineering and Applied Science has been awarded a $10 million grant by the National Science Foundation's Expeditions in Computing program to develop high-performance, energy efficient, customizable computing that could revolutionize the way computers are used in health care and other important applications.

In particular, UCLA Engineering researchers will demonstrate how the new technology, known as domain-specific computing, could transform the role of medical imaging and hemodynamic simulation, providing more cost-effective and convenient solutions for preventive, diagnostic and therapeutic procedures and dramatically improving health care quality, efficiency and patient outcomes.

"This significant award is another testament to the world-class faculty here at UCLA who continue to push the envelope to solve society's most pressing issues," said UCLA Chancellor Gene Block. "We are grateful to the NSF, which has repeatedly provided crucial funding to our faculty, helping to place the university among the nation's top five in research funding."

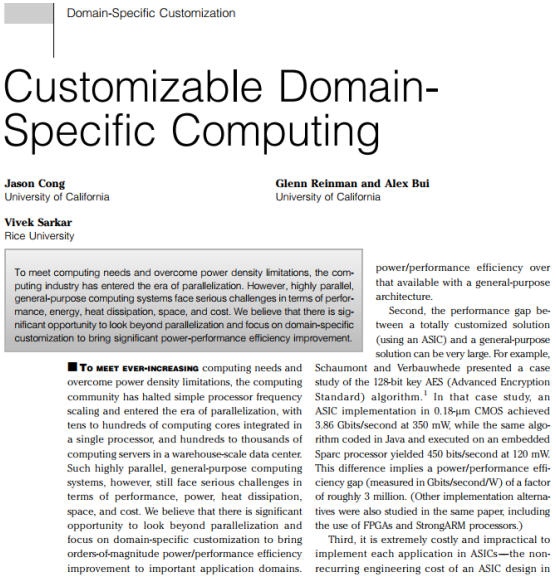
In an effort to meet ever-increasing computing needs in various fields, the computing industry has entered an "era of parallelization," in which tens of thousands of computer servers are connected in warehouse-scale data centers, said Jason Cong, the Chancellor's Professor of Computer Science and director of the new UCLA Center for Domain-Specific Computing (CDSC), which will oversee the research. But these parallel, general-purpose computing systems still face serious challenges in terms of performance, energy, space and cost.

Domain-specific computing holds significant advantages, Cong said. While general-purpose computing relies on computer architecture and languages aimed at any type of application, domain-specific computing utilizes a customizable architecture and custom-oriented, high-level computer languages tailored to a particular application area or domain — in this case, medical imaging and hemodynamic modeling. This customization ultimately results in much less energy consumption, faster results, lower costs and increased productivity.

The goal of the new UCLA center, Cong said, is to look beyond parallelization and focus on domain-specific customization to bring significant power-performance efficiency improvement to important application domains.

… to look beyond parallelization and focus on domain-specific customization to bring significant power-performance efficiency …

Customized Computing has been Our Research Focus Since 2009



Domain-Specific Customization

Customizable Domain-Specific Computing

Jason Cong
University of California

Glenn Reinman and Alex Bui
University of California

Vivek Sarkar
Rice University

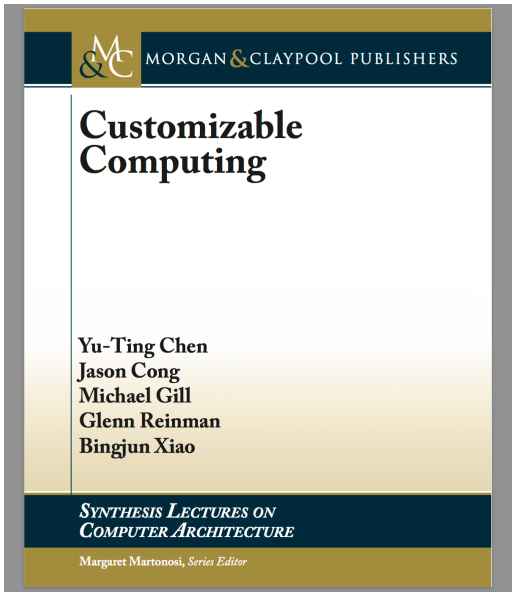
To meet computing needs and overcome power density limitations, the computing industry has entered the era of parallelization. However, highly parallel, general-purpose computing systems face serious challenges in terms of performance, energy, heat dissipation, space, and cost. We believe that there is significant opportunity to look beyond parallelization and focus on domain-specific customization to bring significant power-performance efficiency improvement.

■ To meet ever-increasing computing needs and overcome power density limitations, the computing community has halted simple processor frequency scaling and entered the era of parallelization, with tens to hundreds of computing cores integrated in a single processor, and hundreds to thousands of computing servers in a warehouse-scale data center. Such highly parallel, general-purpose computing systems, however, still face serious challenges in terms of performance, power, heat dissipation, space, and cost. We believe that there is significant opportunity to look beyond parallelization and focus on domain-specific customization to bring orders-of-magnitude power/performance efficiency improvement to important application domains. power/performance efficiency over that available with a general-purpose architecture.

Second, the performance gap between a totally customized solution (using an ASIC) and a general-purpose solution can be very large. For example, Schaumont and Verbauwhede presented a case study of the 128-bit key AES (Advanced Encryption Standard) algorithm.[1] In that case study, an ASIC implementation in 0.18-μm CMOS achieved 3.86 Gbits/second at 350 mW, while the same algorithm coded in Java and executed on an embedded Sparc processor yielded 450 bits/second at 120 mW. This difference implies a power/performance efficiency gap (measured in Gbits/second/W) of a factor of roughly 3 million. (Other implementation alternatives were also studied in the same paper, including the use of FPGAs and StrongARM processors.)

Third, it is extremely costly and impractical to implement each application in ASICs—the nonrecurring engineering cost of an ASIC design in

IEEE Design & Test, 2011



MORGAN & CLAYPOOL PUBLISHERS

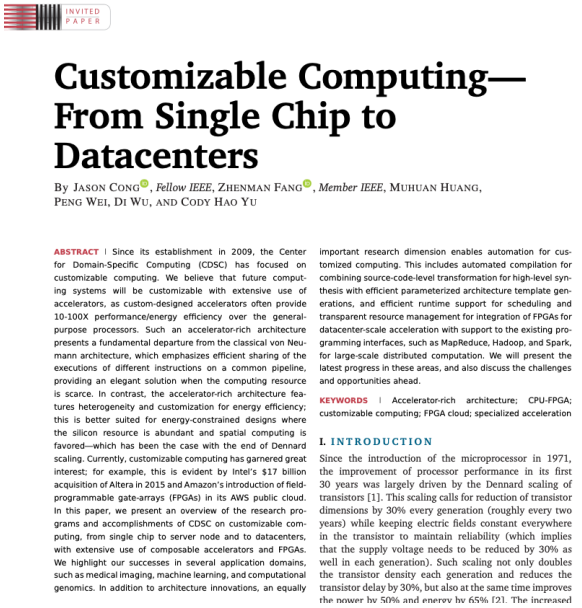
Customizable Computing

Yu-Ting Chen
Jason Cong
Michael Gill
Glenn Reinman
Bingjun Xiao

Synthesis Lectures on Computer Architecture

Margaret Martonosi, *Series Editor*

Synthesis Lectures on Computer Architectures, 2015



INVITED PAPER

Customizable Computing—From Single Chip to Datacenters

By Jason Cong, *Fellow IEEE*, Zhenman Fang, *Member IEEE*, Muhuan Huang, Peng Wei, Di Wu, and Cody Hao Yu

ABSTRACT | Since its establishment in 2009, the Center for Domain-Specific Computing (CDSC) has focused on customizable computing. We believe that future computing systems will be customizable with extensive use of accelerators, as custom-designed accelerators often provide 10-100X performance/energy efficiency over the general-purpose processors. Such an accelerator-rich architecture presents a fundamental departure from the classical von Neumann architecture, which emphasizes efficient sharing of the executions of different instructions on a common pipeline, providing an elegant solution when the computing resource is scarce. In contrast, the accelerator-rich architecture features heterogeneity and customization for energy efficiency; this is better suited for energy-constrained designs where the silicon resource is abundant and spatial computing is favored—which has been the case with the end of Dennard scaling. Currently, customizable computing has garnered great interest; for example, this is evident by Intel's $17 billion acquisition of Altera in 2015 and Amazon's introduction of field-programmable gate-arrays (FPGAs) in its AWS public cloud. In this paper, we present an overview of the research programs and accomplishments of CDSC on customizable computing, from single chip to server node and to datacenters, with extensive use of composable accelerators and FPGAs. We highlight our successes in several application domains, such as medical imaging, machine learning, and computational genomics. In addition to architecture innovations, an equally important research dimension enables automation for customized computing. This includes automated compilation for combining source-code-level transformation for high-level synthesis with efficient parameterized architecture template generations, and efficient runtime support for scheduling and transparent resource management for integration of FPGAs for datacenter-scale acceleration with support to the existing programming interfaces, such as MapReduce, Hadoop, and Spark, for large-scale distributed computation. We will present the latest progress in these areas, and also discuss the challenges and opportunities ahead.

KEYWORDS | Accelerator-rich architecture; CPU-FPGA; customizable computing; FPGA cloud; specialized acceleration

I. INTRODUCTION

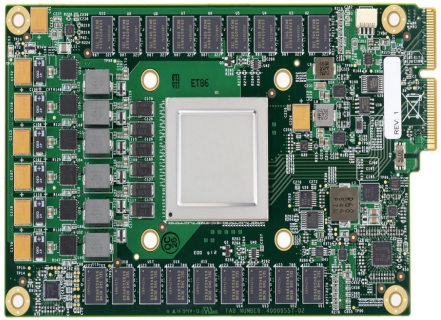
Since the introduction of the microprocessor in 1971, the improvement of processor performance in its first 30 years was largely driven by the Dennard scaling of transistors [1]. This scaling calls for reduction of transistor dimensions by 30% every generation (roughly every two years) while keeping electric fields constant everywhere in the transistor to maintain reliability (which implies that the supply voltage needs to be reduced by 30% as well in each generation). Such scaling not only doubles the transistor density each generation and reduces the transistor delay by 30%, but also at the same time improves the power by 50% and energy by 65% [2]. The increased

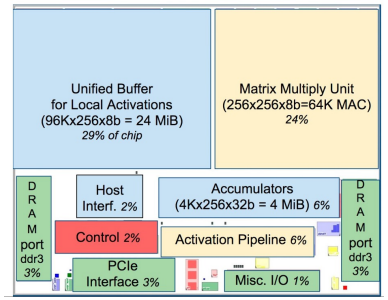
Proceedings of IEEE, 2019

Successful Examples of Customization

- Example:
  - Google TPU (Tensor Processing Unit)
- First version: 2014
- Revised TPU (2017), for training and inference
  - DRAM, 2 DDR3 -> GDDR5, 34GB/s -> 180GB/s
  - 200x perf/W of Haswell CPU, 70x perf/W of K80 GPU

  Based on data in [ISCA2017]
- Limitations:
  - Too costly for individuals (or small companies) to design
  - Take too much time to build



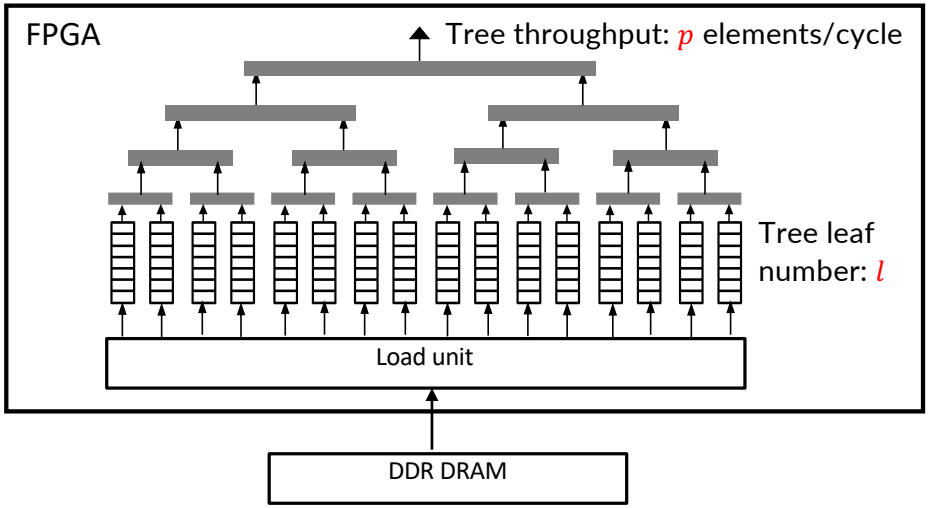


331 mm2 (< CPU: Haswell 18 core, 662 mm2)
75 TDP Watts (< CPU: 145)
91.8 Peak TOPS/chip (CPU: 2.6 TOPS)

Google TPU: In-Datacenter Performance Analysis of a Tensor Processing Unit, ISCA 2017

Customized Computing on FPGAs: Example: Scalable Sorting [ISCA 2020]

- Bonsai: Adaptive merge tree sort solution (compute and I/O optimal)
  - Optimized configuration of merge sort kernel for different memory configurations
  - Best DRAM-scale sorting performance
  - Scale to TB sorting via reconfiguration



DRAM bandwidth	Tree throughput p	Tree leaf number l
16 GB/s	8	128
32 GB/s	16	128
64 GB/s	32	64

phase 1: 4 trees

switch to phase

SSD/FLASH I/O

Optimal configuration for different available DRAM bandwidth when sorting 32-bit integers

I/O FPGA

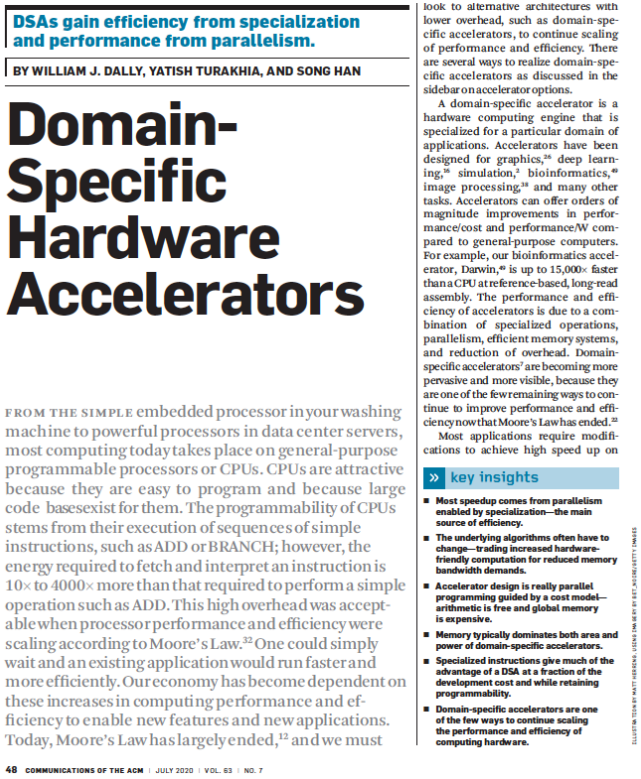
phase 2: 1 many-leaves tree ($p = 8, l = 256$)

Use of FPGAs trade-off performance for design cost, flexibility, and time-to-silicon

Power of Customization (Domain-Specific Accelerators)

- Special **Data Types** and **Operations**
  - Do in 1 cycle what normally takes 10s or 100s – **10-1000x efficiency gain**

Most significant on ASIC (if one can afford cost and time)
Still very substantial speedup on FPGAs despite its overhead



DSAs gain efficiency from specialization and performance from parallelism.

BY WILLIAM J. DALLY, YATISH TURAKHIA, AND SONG HAN

Domain-Specific Hardware Accelerators

FROM THE SIMPLE embedded processor in your washing machine to powerful processors in data center servers, most computing today takes place on general-purpose programmable processors or CPUs. CPUs are attractive because they are easy to program and because large code bases exist for them. The programmability of CPUs stems from their execution of sequences of simple instructions, such as ADD or BRANCH; however, the energy required to fetch and interpret an instruction is 10× to 4000× more than that required to perform a simple operation such as ADD. This high overhead was acceptable when processor performance and efficiency were scaling according to Moore's Law.[32] One could simply wait and an existing application would run faster and more efficiently. Our economy has become dependent on these increases in computing performance and efficiency to enable new features and new applications. Today, Moore's Law has largely ended,[12] and we must look to alternative architectures with lower overhead, such as domain-specific accelerators, to continue scaling of performance and efficiency. There are several ways to realize domain-specific accelerators as discussed in the sidebar on accelerator options.

A domain-specific accelerator is a hardware computing engine that is specialized for a particular domain of applications. Accelerators have been designed for graphics,[26] deep learning,[16] simulation,[2] bioinformatics,[49] image processing,[18] and many other tasks. Accelerators can offer orders of magnitude improvements in performance/cost and performance/W compared to general-purpose computers. For example, our bioinformatics accelerator, Darwin,[49] is up to 15,000× faster than a CPU at reference-based, long-read assembly. The performance and efficiency of accelerators is due to a combination of specialized operations, parallelism, efficient memory systems, and reduction of overhead. Domain-specific accelerators' are becoming more pervasive and more visible, because they are one of the few remaining ways to continue to improve performance and efficiency now that Moore's Law has ended.[22]

Most applications require modifications to achieve high speed up on

» key insights

- Most speedup comes from parallelism enabled by specialization—the main source of efficiency.
- The underlying algorithms often have to change—trading increased hardware-friendly computation for reduced memory bandwidth demands.
- Accelerator design is really parallel programming guided by a cost model—arithmetic is free and global memory is expensive.
- Memory typically dominates both area and power of domain-specific accelerators.
- Specialized instructions give much of the advantage of a DSA at a fraction of the development cost and while retaining programmability.
- Domain-specific accelerators are one of the few ways to continue scaling the performance and efficiency of computing hardware.

ILLUSTRATION BY MATT HERSEY; LEAD IMAGE BY BET_ARCHIVE/GETTY IMAGES

48 COMMUNICATIONS OF THE ACM | JULY 2020 | VOL. 63 | NO. 7

CACM July 2020

Question:

Can Every Programmer Easily Design DSAs?

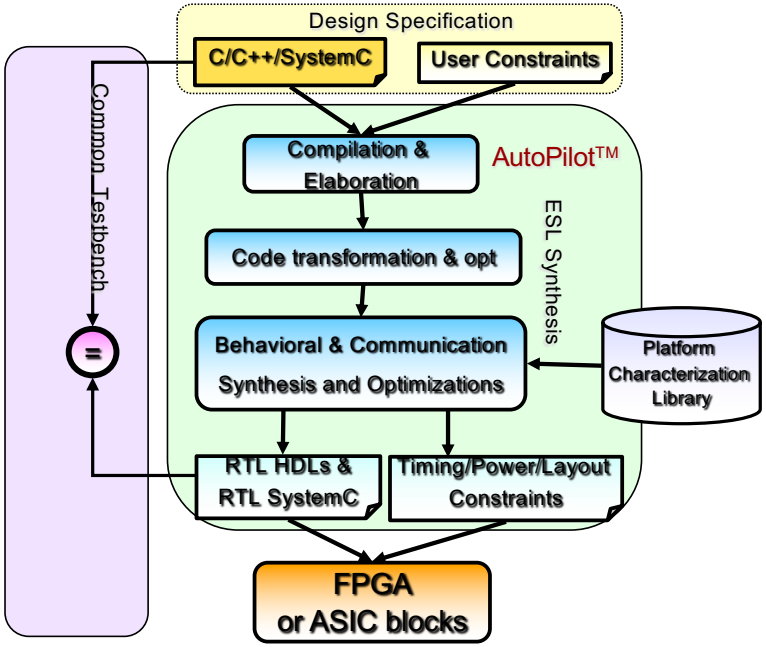
Or

Can Every Serious Programmer Easily Design DSAs?

Current Answer: Yes and No

It's Natural to Think about High-Level Synthesis (HLS)?

Significant progress in the past decade



- Example: xPilot (UCLA 2006) -> AutoPilot (AutoESL) -> Vivado HLS (Xilinx 2011-)
  - Platform-based C to RTL synthesis
  - Synthesize pure ANSI-C and C++, GCC-compatible compilation flow leveraging LLVM framework
  - Full support of IEEE-754 floating point data types & operations
  - Efficiently handle bit-accurate fixed-point arithmetic
  - SDC-based scheduling
  - Automatic memory partitioning
- …

QoR matches or exceeds manual RTL for many designs

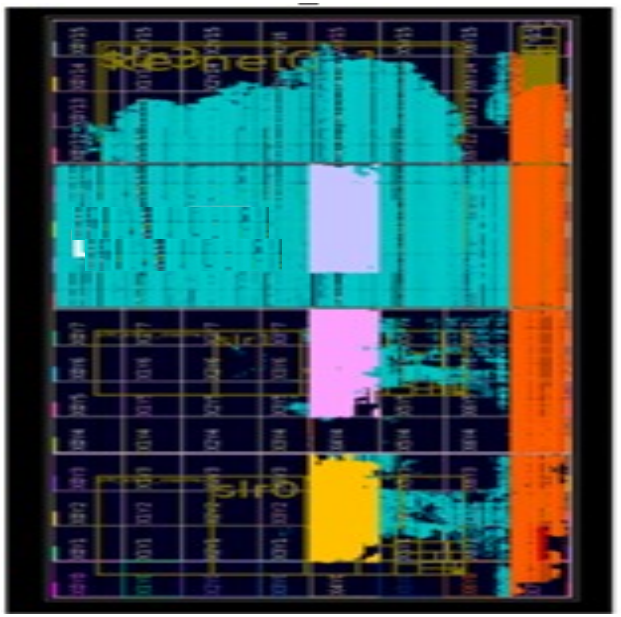
TCAD April 2011 (keynote paper) "High-Level Synthesis for FPGAs: From Prototyping to Deployment"

Good News: Not Difficult to Create Circuits from C/C++ Using HLS

Example code

- MVT kernel from Polybench
  - Two matrix-vetor multiplications

```
void kernel_mvt(double x1[120], double x2[120],
                double y_1[120], double y_2[120],
                double A[120][120]) {


 for (int i = 0; i < 120; i++) {
    for (int j = 0; j < 120; j++) {
      x1[i] += A[i][j] * y_1[j];
   }
 }

 for (int i = 0; i < 120; i++) {
   for (int j = 0; j < 120; j++) {
      x2[i] += A[j][i] * y_2[j];
  }
 }
}
```



Challenge 1: Synthesized Circuit May Not Have Good Performance

- Not surprising if you have done multi-core programming – the same problem!
- Need to add *pragmas (microarchitecture hints).*

Example code: MVT kernel from Polybench

```
void kernel_mvt(double x1[120], double x2[120],
                double y_1[120], double y_2[120],
                double A[120][120]) {


 for (int i = 0; i < 120; i++) {
    for (int j = 0; j < 120; j++) {
      x1[i] += A[i][j] * y_1[j];
   }
  }

 for (int i = 0; i < 120; i++) {
    for (int j = 0; j < 120; j++) {
      x2[i] += A[j][i] * y_2[j];
   }
  }
}
```

When targeting FPGA, 13x slower than running on a single-core CPU

122x speedup

After proper pragma insertions

```
void kernel_mvt(double x1[120], double x2[120],
                double y_1[120], double y_2[120],
                double A[120][120]) {


#pragma ACCEL PIPELINE flatten
for (int i = 0; i < 120; i++) {
  for (int j = 0; j < 120; j++) {
     x1[i] += A[i][j] * y_1[j];
 } }

#pragma ACCEL PARALLEL FACTOR=15
  for (int i = 0; i < 120; i++) {
#pragma ACCEL PARALLEL reduction = x2 FACTOR=12
    for (int j = 0; j < 120; j++) {
      x2[i] += A[j][i] * y_2[j];
 } }
}
```

Based on the Merlin Compiler, open-sourced by AMD/Xilinx

Challenge 2: # Possibilities for Pragmas Insertion Can Be Very Large!

Example code: MVT kernel from Polybench

```
void kernel_mvt(double x1[120], double x2[120],
                double y_1[120], double y_2[120],
                double A[120][120]) {


 for (int i = 0; i < 120; i++) {
    for (int j = 0; j < 120; j++) {
      x1[i] += A[i][j] * y_1[j];
   }
  }

 for (int i = 0; i < 120; i++) {
   for (int j = 0; j < 120; j++) {
      x2[i] += A[j][i] * y_2[j];
   }
  }
}
```

When targeting FPGA, 13x slower than running on a single-core CPU

122x speedup

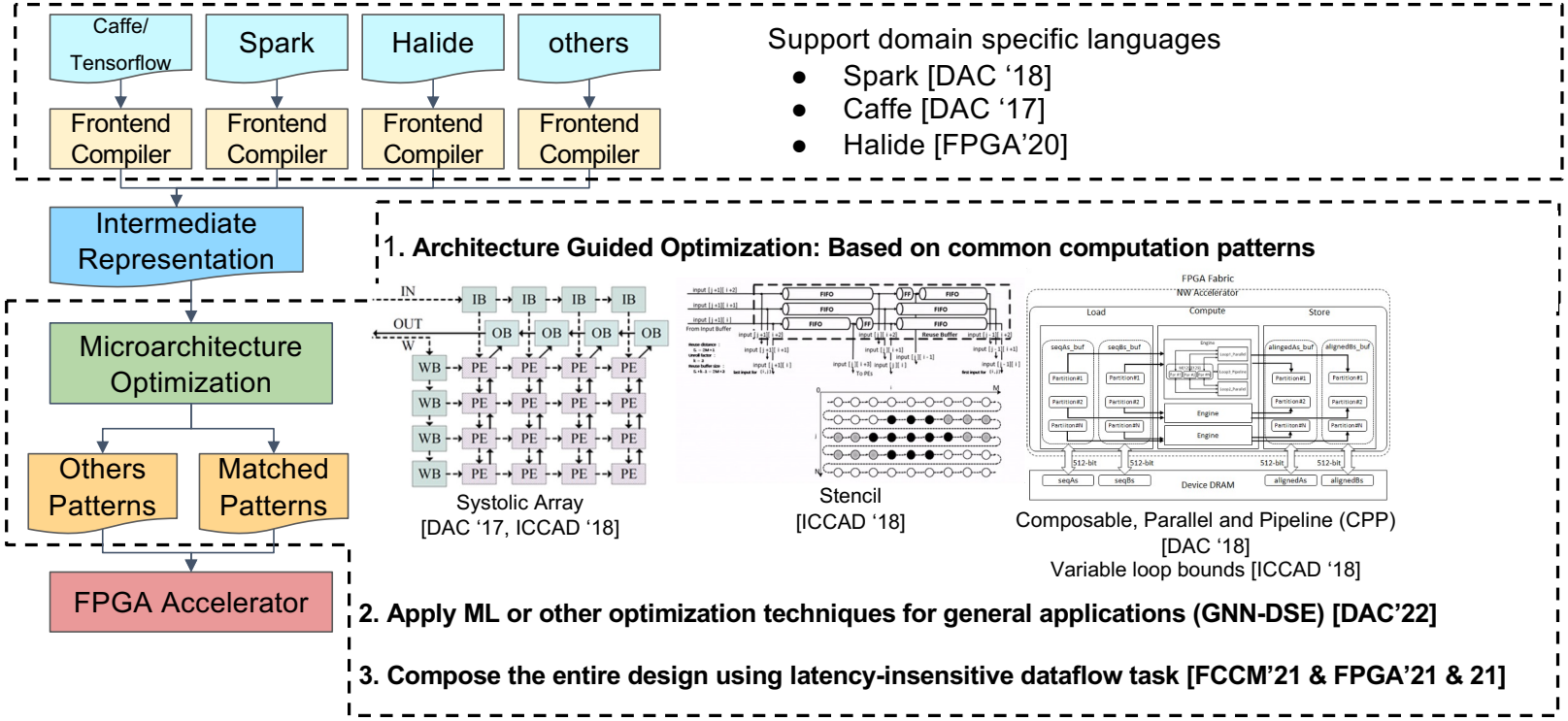
Solution space

- > 3M design choices

```
void kernel_mvt(double x1[120], double x2[120],
                double y_1[120], double y_2[120],
                double A[120][120]) {
#pragma ACCEL PIPELINE auto{__PIPE__L0}
#pragma ACCEL TILE FACTOR=auto{__TILE__L0}
#pragma ACCEL PARALLEL FACTOR=auto{__PARA__L0}
for (int i = 0; i < 120; i++) {
#pragma ACCEL PIPELINE auto{__PIPE__L2}
#pragma ACCEL PARALLEL FACTOR=auto{__PARA__L2}
  for (int j = 0; j < 120; j++) {
    x1[i] += A[i][j] * y_1[j];  } }
#pragma ACCEL PARALLEL FACTOR=15
#pragma ACCEL PIPELINE auto{__PIPE__L1}
#pragma ACCEL TILE FACTOR=auto{__TILE__L1}
#pragma ACCEL PARALLEL FACTOR=12
#pragma ACCEL PARALLEL FACTOR=auto{__PARA__L1}
 for (int j = 0; j < 120; j++) {
   for (int i = 0; i < 120; i++) {
     x2[i] += A[j][i] * y_2[j];  } }
#pragma ACCEL PIPELINE auto{__PIPE__L3}
#pragma ACCEL PARALLEL FACTOR=auto{__PARA__L3}
  for (int j = 0; j < 120; j++) {
    x2[i] += A[j][i] * y_2[j];  } } }
```

Search space by AutoDSE

Overview of Our Approach



Goal: "Democratize" accelerator designs for customized computing

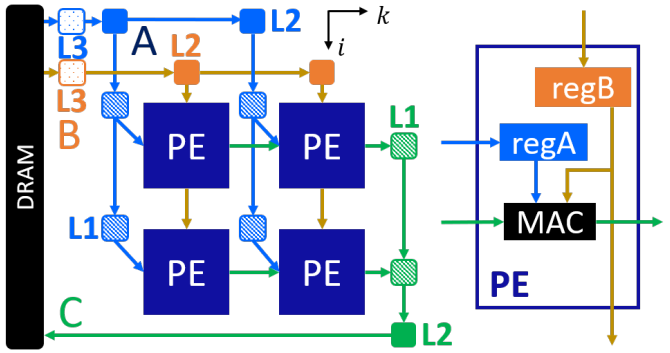
Example of Architecture-Guided Optimization: AutoSA

Wang, Jie, Licheng Guo, and Jason Cong. "AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA." FPGA'2021

```
#pragma scop
for (int i = 0; i < M; ++i)
    for (int j = 0; j < N; ++j) {
S0:     C[i][j] = 0;
        for (int k = 0; k < K; ++k)
S1:         C[i][j] += A[i][k] * B[k][j];
    }
#pragma endscop
```

Input: C code

AutoSA



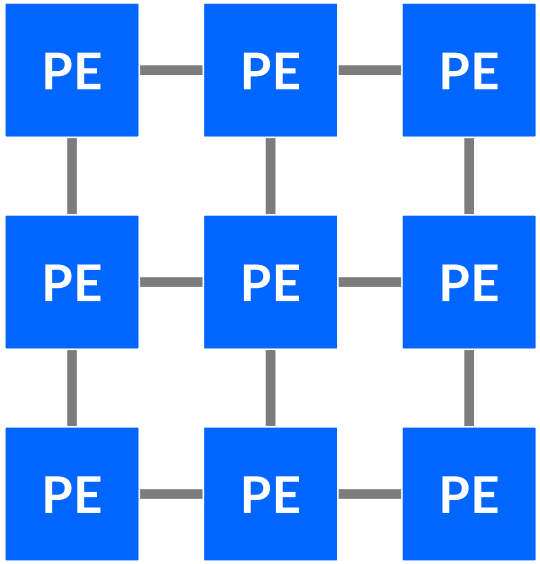
Output: Systolic array design in HLS C

Systolic Array Advantages

Parallelism

↓

Performance

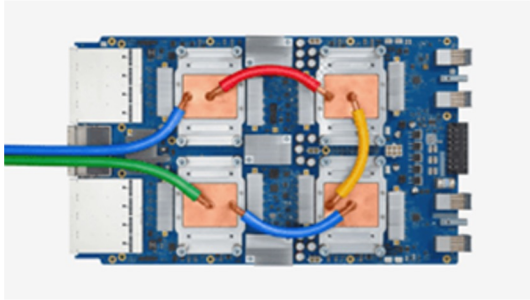


Locality

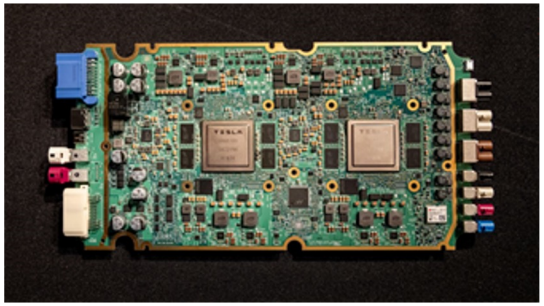
↓

Energy Efficiency

Many Accelerators are Based on Systolic Arrays



Google TPU



Tesla Self-Driving Chip

Amazon Infrentia

Systolic Array Design Stories from Industry

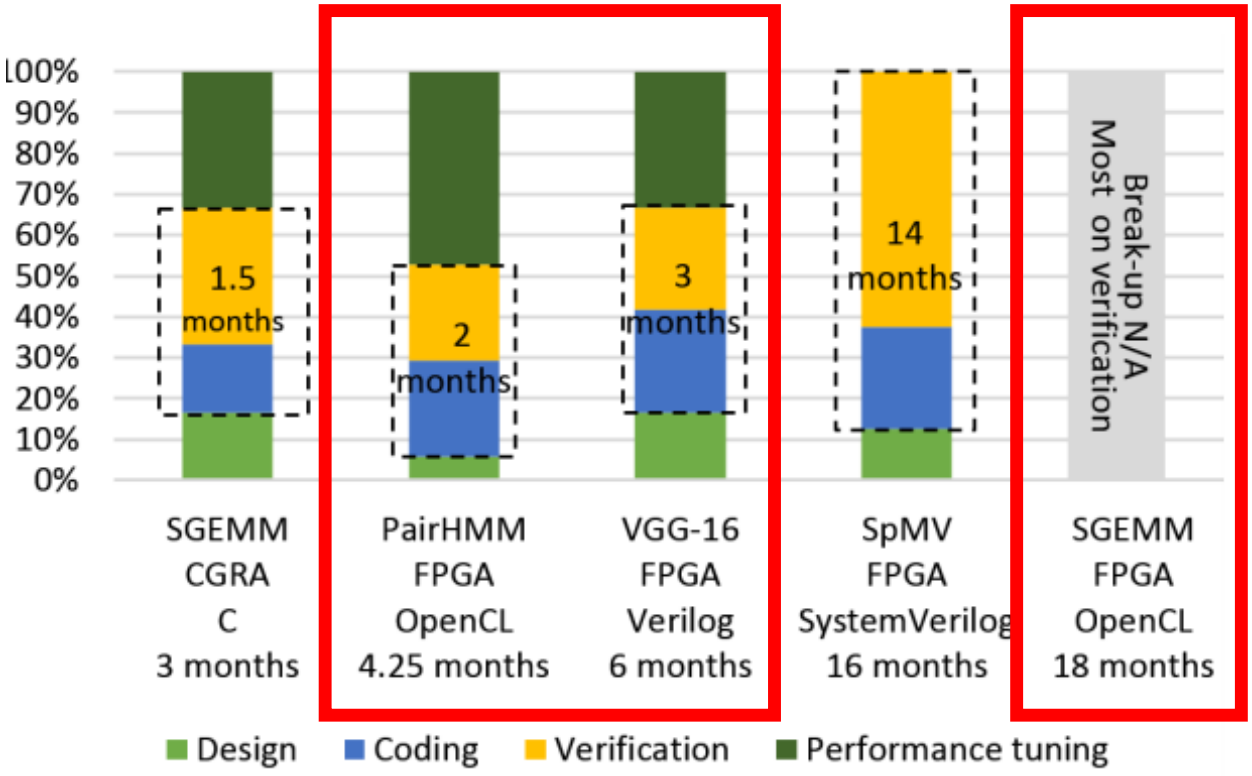
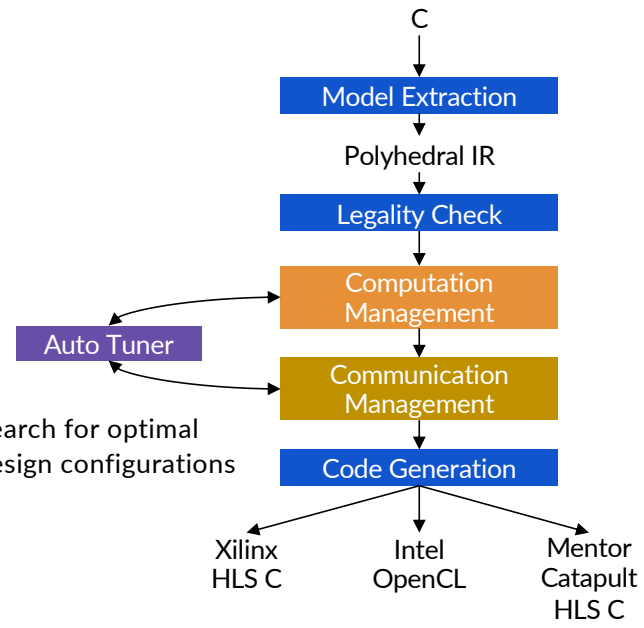


 Figure source: Hongbo Rong, "*Programmatic control of a compiler for generating high-performance spatial hardware*", arXiv preprint arXiv:1711.07606 (2017).

Overview of AutoSA Compilation Flow

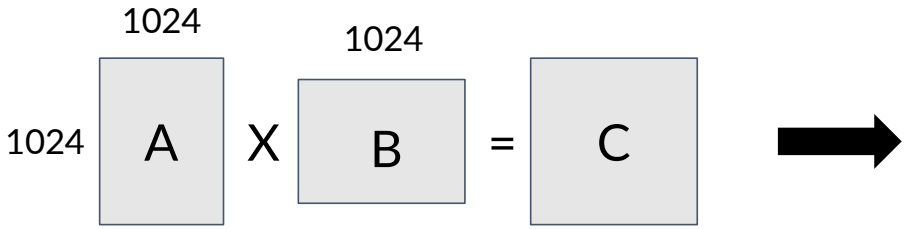


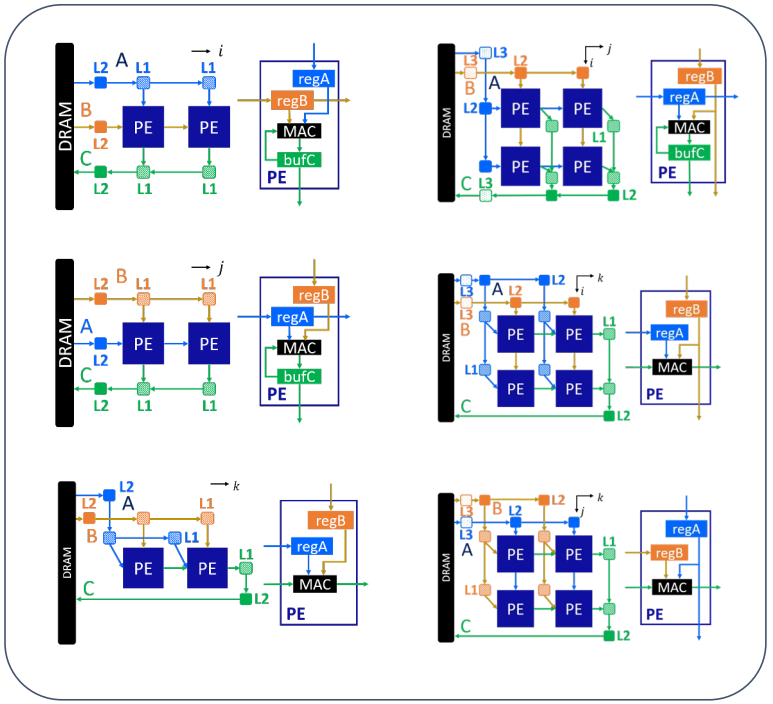
- Search for optimal design configurations

- Extract polyhedral model from the source code.
- Examine if the target program can be mapped to systolic array.
- Construct and optimize PE arrays
  - Space-time mapping, array partitioning, latency hiding, vectorization
- Construct and optimize I/O network
  - I/O network analysis, double buffering, data-packing
- Generate target hardware code

Challenge: Large Design Space & Many Optimization Opportunities

Example: Matrix Multiplication





Dataflows types(6) X Dataflow Configurations(O(2^40))

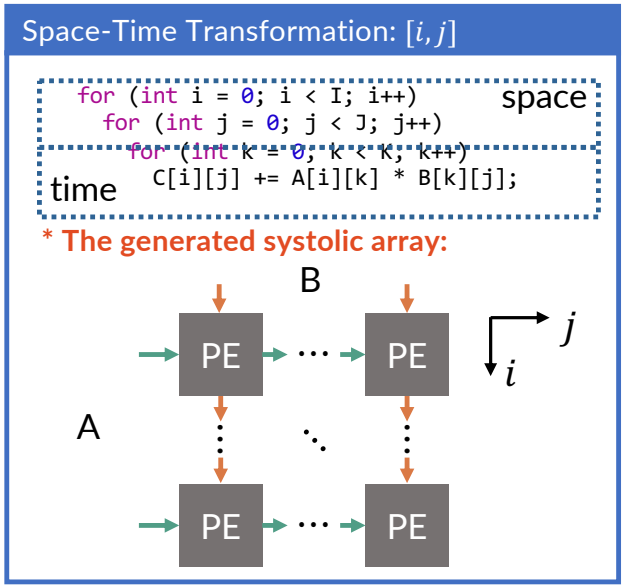
A Closer Look at Computation Management

- **Space-time mapping**: transforming the program to a systolic array with space-time mapping.

Input Code of MM:

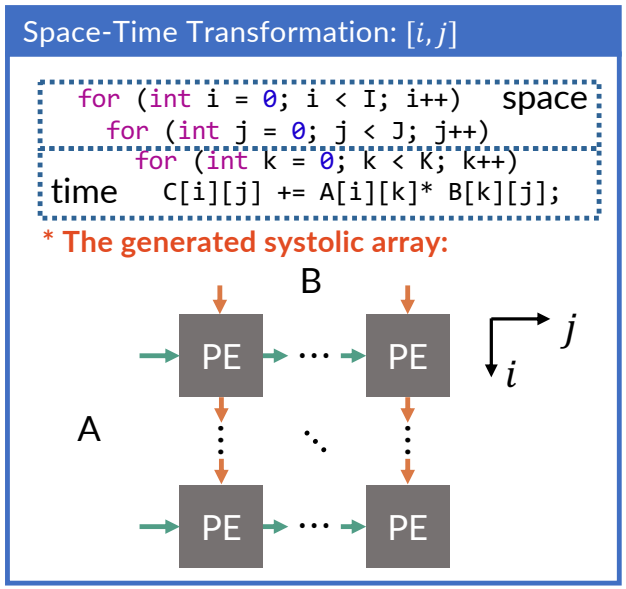
```
for (int i = 0; i < I; i++)
  for (int j = 0; j < J; j++)
    for (int k = 0; k < K; k++)
      C[i][j] += A[i][k] * B[k][j];
```

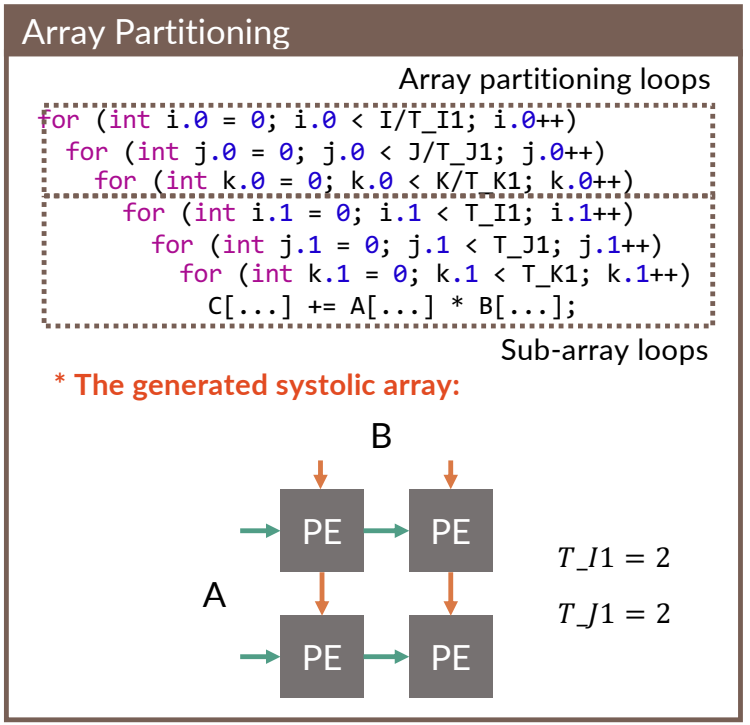
Note: Initialization of C omitted for brevity.



A Closer Look at Computation Management

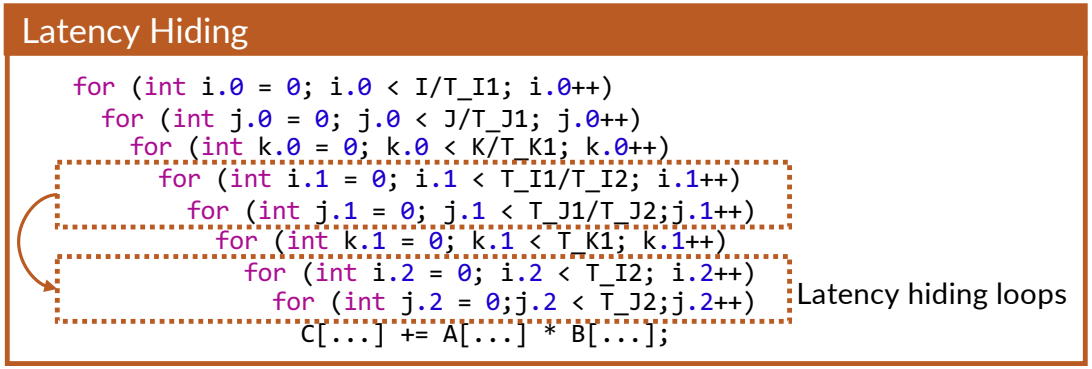
- **Array partitioning**: partitioning the array into smaller sub-arrays to fit limited on-chip resource.





A Closer Look at Computation Management

- **Latency hiding**: permuting parallel loops inside to hide computation latency.



A Closer Look at Computation Management

- **SIMD vectorization**: vectorizing computation to amortize the PE control overheads.

SIMD Vectorization

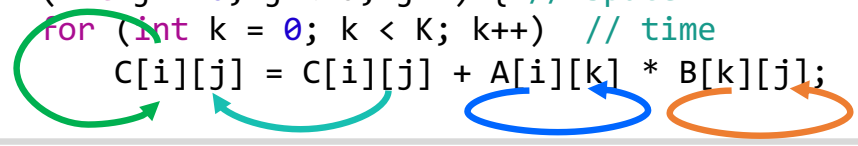
```
for (int i.0 = 0; i.0 < I/T_I1; i.0++)
  for (int j.0 = 0; j.0 < J/T_J1; j.0++)
    for (int k.0 = 0; k.0 < K/T_K1; k.0++)
      for (int i.1 = 0; i.1 < T_I1/T_I2; i.1++)
        for (int j.1 = 0; j.1 < T_J1/T_J2; j.1++)
          for (int k.1 = 0; k.1 < T_K1/T_K2; k.1++)
            for (int i.2 = 0; i.2 < T_I2; i.2++)
              for (int j.2 = 0; j.2 < T_J2; j.2++)
                for (int k.2 = 0; k.2 < T_K2;k.2++)
                  C[...] += A[...] * B[...];
```

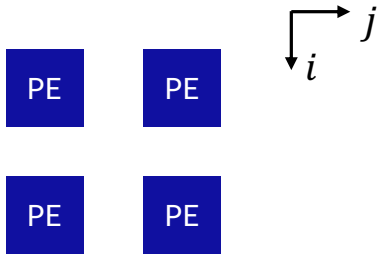
SIMD loop

What about Data Communication?

- Polyhedral model supports precise data dependence analysis.

```
for (int i = 0; i < I; i++) // space
    for (int j = 0; j < J; j++) { // space
        for (int k = 0; k < K; k++)  // time
S1:         C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }
```



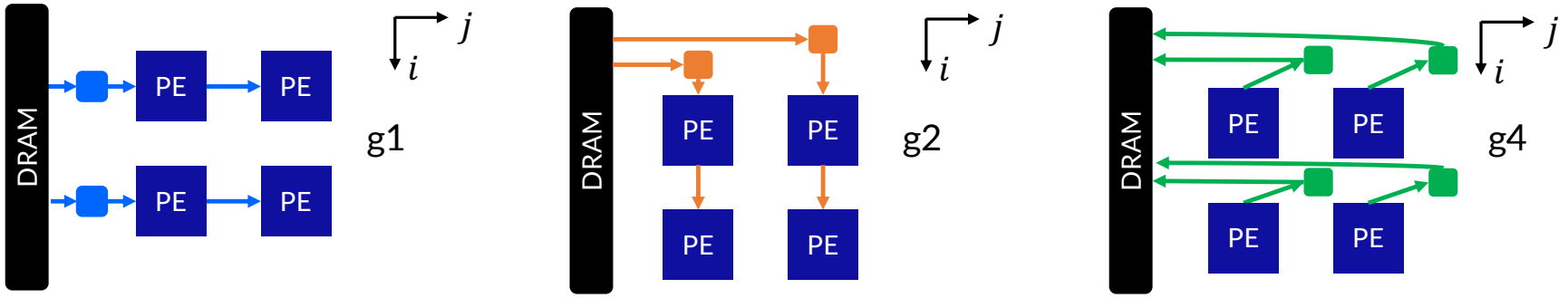


Dependence	Dependence Type	Array Access	Dependence Distance	I/O Group
D1	Read (RAR)	A[i][k]	$(\mathbf{0},\mathbf{1},0)$	g1
D2	Read (RAR)	B[k][j]	$(\mathbf{1},\mathbf{0},0)$	g2
D3	Flow (RAW)	C[i][j]	$(\mathbf{0},\mathbf{0},1)$	g3
D4	Output (WAW)	C[i][j]	$(\mathbf{0},\mathbf{0},1)$	g4

We omit the statement of array initialization for brevity.

Use Dependency to Construct Communication Network

- Polyhedral model supports precise data dependence analysis.

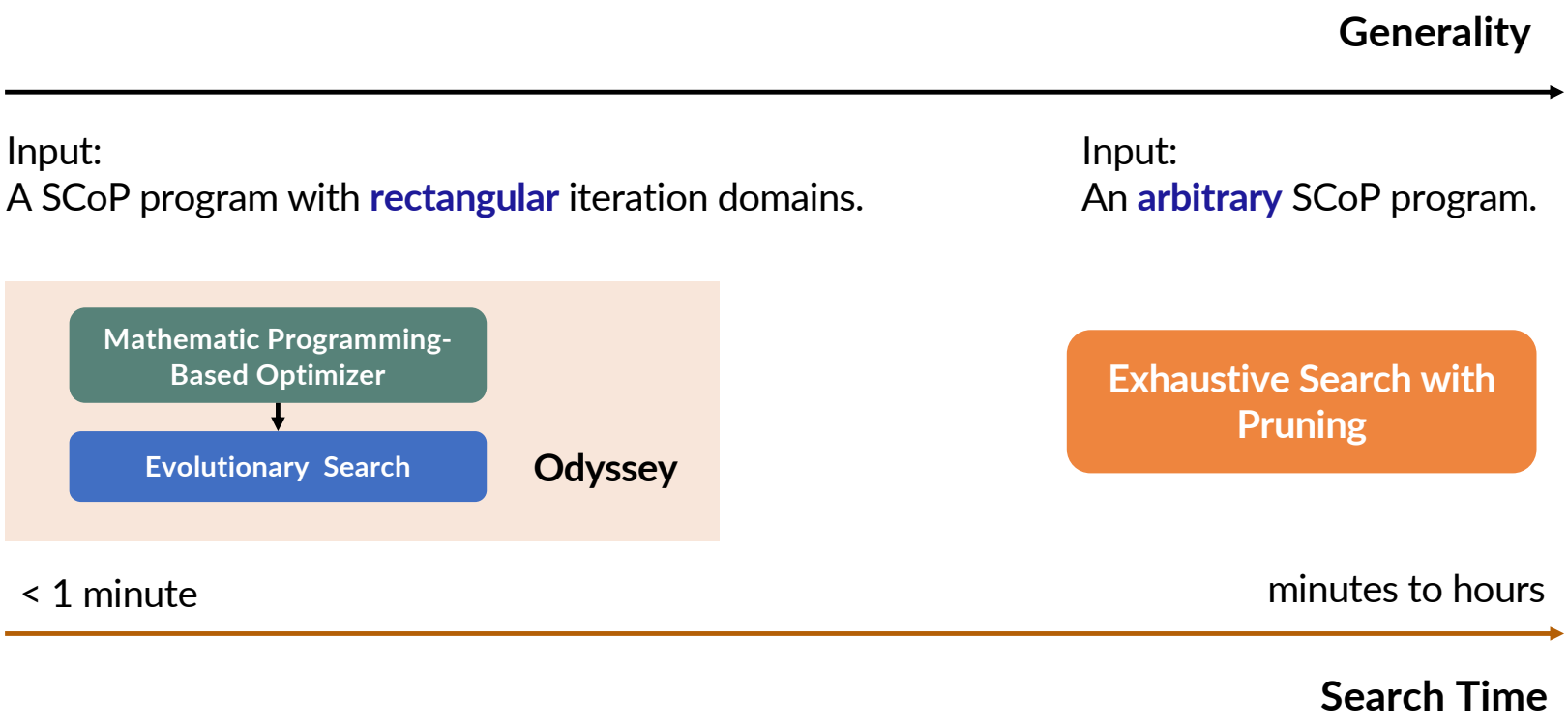


Dependence	Dependence Type	Array Access	Dependence Distance	I/O Group
D1	Read (RAR)	`A[i][k]`	$(\mathbf{0},\mathbf{1},0)$	g1
D2	Read (RAR)	`B[k][j]`	$(\mathbf{1},\mathbf{0},0)$	g2
D3	Flow (RAW)	`C[i][j]`	$(\mathbf{0},\mathbf{0},1)$	g3
D4	Output (WAW)	`C[i][j]`	$(\mathbf{0},\mathbf{0},1)$	g4

We omit the statement of array initialization for brevity.

Example: I/O network generation based on the polyhedral model

Auto-Tuning in AutoSA (More in Late Slides)



Benchmark Examples and Productivity Gain

Application	Problem Size	#Statements	Input C LOC	Output HLS LOC
Matrix Multiplication	$[i, j, k]$: [1024,1024,1024]	2	7	9265
CNN	$[i, o, h, w, p, q]$: [512,512,56,56,3,3]	2	10	9861
MTTKRP	$[i, k, l, j]$: [512,512,512,512]	2	9	7858
TTMc	$[i, j, k, l, m]$: [128,128,128,128,128]	2	9	7637
LU Decomposition	$[n]$: [12/16/20/24]	9	27	1316

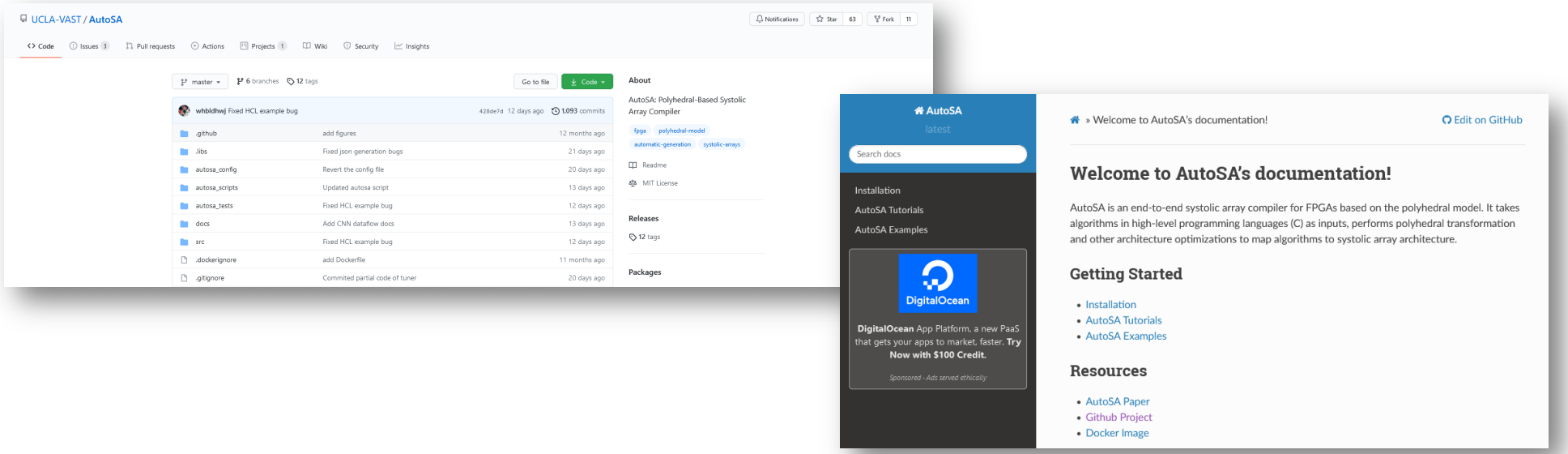
Complex systolic Array from C-to-Silicon in a day!
Recall that common industry practice requires 4-18 months.

Performance

Benchmark		Platform	Array Sizes	Data Type	GFLOPs	MHz	DSP Efficiency
CNN	Wei et al. '17	Intel Arria 10	8x19x8	FP32	602.8	253	97%
	AutoSA	Xilinx Alveo U250	16x14x8	FP32	**950.2**	**272**	**97%**
MTTKRP	Srivastava et al. '19	Intel Arria 10	8x9x16	FP32	700	204	99%
	AutoSA	Xilinx Alveo U250	16x8x8	FP32	**896.7**	**296**	**99%**
TTMc	Srivastava et al. '19	Intel Arria 10	8x10x16	FP32	738	205	94%
	AutoSA	Xilinx Alveo U250	16x8x8	FP32	**886.2**	**290**	**99%**

AutoSA is Open-Sourced

- Github: https://github.com/UCLA-VAST/AutoSA
- Document: https://autosa.readthedocs.io/en/latest/

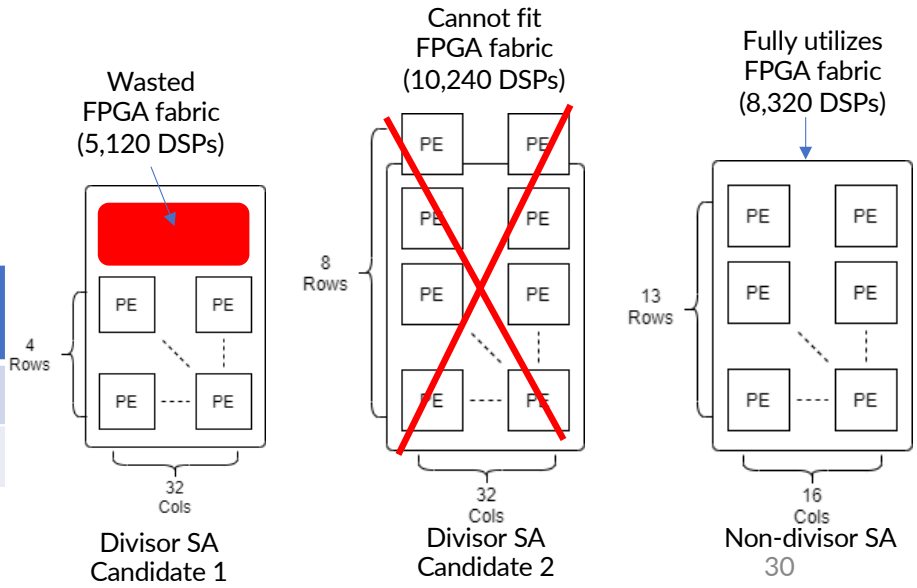


Some Architecture Insights from AutoSA

- Example: 1024x1024x1024 GEMM
- Common wisdom: dimensions of a systolic array be divisors of the problem size.
  - Timeloop ISPASS '19 (MIT, Nvidia, Stanford),
  - dMazeRunner TECS '19 (Ariazon State Univ., Yonsei Univ., Intel)
  - Interstellar ASPLOS '20 (Stanford, Tsinghua)
- Non-divisor solution can be 50% faster

SA Size (Cols,Rows, SIMD)	DSPs	Frequency	Throughput GFLOP/s
32x4x8	5120	257 MHz	506.71
16x13x8	8320	243 MHz	764.46

Non divisor



Some More Architecture Insight using AutoSA

- Common wisdom: Minimize off-chip communication. E.g
  - Marvel Arxiv '20 (Georgia Tech, Nvidia),
  - Chen et al. HPCA '20 (UCAS, Tsinghua Univ.)
- Again, not necessarily!

SA Size (Cols,Rows, SIMD)	Minimization Goal	DSPs	Frequency	Throughput	DRAM Traffic	CTC (FLOP/byte)	Effective Bandwidth
32x4x8	DRAM Traffic	5120	282 MHz	496.16 GFLOP/s	16.7 MB	128	4.3 GB/s
16x13x8	Latency	8320	243 MHz	764.46 GFLOP/s	80.3 MB	26.7	36.5 GB/s

What about General C/C++ Programs?

- Adopting the Merlin Compiler, developed by Falcon Computing (acquired by Xilinx in 2020 and open-sourced in 2021)

`#pragma ACCEL parallel`

- Run multiple loop iterations in parallel (instruction/task-level)

`#pragma ACCEL pipeline`

- Run multiple loop iterations in pipeline (instruction/task-level)

OpenMP for multi-core CPUs

```
#pragma omp parallel for num_threads(16)
for (int i = 0; i < N; ++i) {
  c[i] += a[i] * b[i];
}
```

Merlin for FPGAs

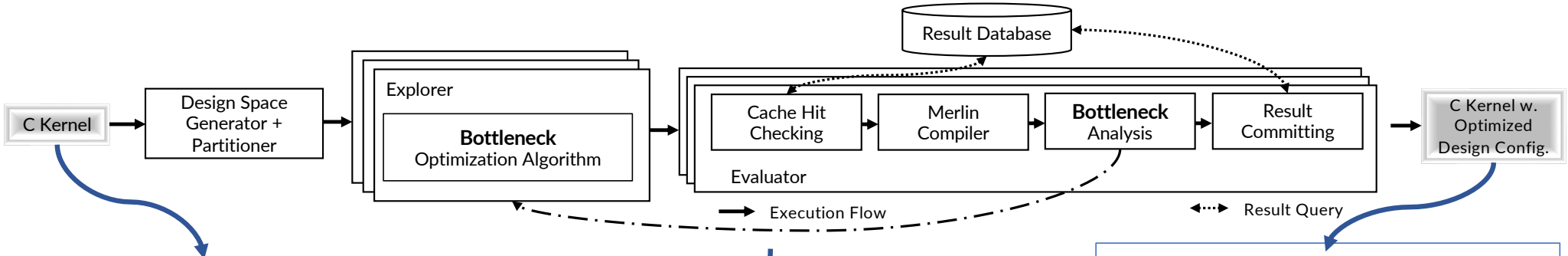
```
#pragma ACCEL parallel factor=16
for (int i = 0; i < N; ++i) {
  c[i] += a[i] * b[i];
}
```

Automated code transformation and transformation

- On-chip memory banking/partitioning/delinearization
- External memory bursting/streaming/coalescing
- Host interface and host code generation (In OpenCL)

Advanced options for parallel: reduction and stencil variables

AutoDSE: Bottleneck-based Optimizer [TODAES'22]



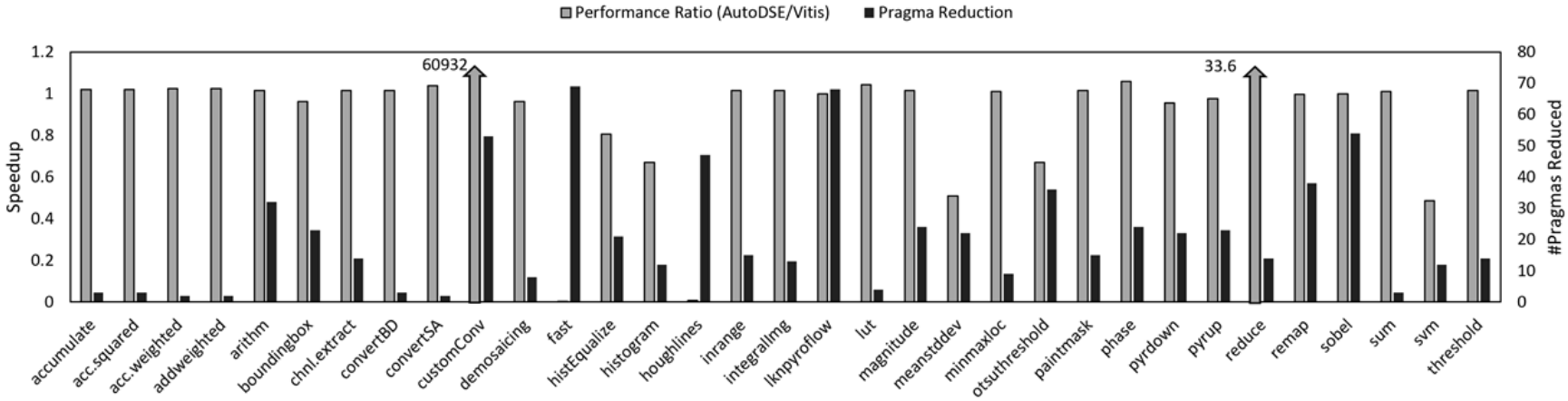

```
void kernel_mvt(double x1[120], double x2[120],
                double y_1[120], double y_2[120],
                double A[120][120]) {


 for (int i = 0; i < 120; i++) {
   for (int j = 0; j < 120; j++) {
     x1[i] += A[i][j] * y_1[j];
   }
 }
 for (int i = 0; i < 120; i++) {
   for (int j = 0; j < 120; j++) {
     x2[i] += A[j][i] * y_2[j];
   }
 }

}
```

```
+-----------------------------------+---+--------------+------+
|             Hierarchy             |TC |      AC      | CPC  |
+-----------------------------------+---+--------------+------+
|kernel_mvt (mvt.c:4)               |   |137701 (100.0%)|137701|
|    auto memory burst for 'x1'(read)|   |   120 (  0.1%)|   120|
|    auto memory burst for 'A'(read) |   |  7200 (  5.2%)|  7200|
|    auto memory burst for 'y_1'(read)|  |   120 (  0.1%)|   120|
|    loop i (mvt.c:15)              |120|   727 (  0.5%)|   727|
|        loop j (mvt.c:18)          |120|             -|     -|
|    auto memory burst for 'y_2'(read)|  |     5 (  0.0%)|     5|
|    auto memory burst for 'x1'(write)|  |   120 (  0.1%)|   120|
|    auto memory burst for 'A'(read) |   |   246 (  0.2%)|   246|
|    auto memory burst for 'x2'(read)|   |    12 (  0.0%)|    12|
|    loop i (mvt.c:28)              |120|127200 ( 92.4%)|127200|
|        loop j (mvt.c:31)          |120|             -|     -|
|    auto memory burst for 'x2'(write)|  |    12 (  0.0%)|    12|
+-----------------------------------+---+--------------+------+
```

```
void kernel_mvt(double x1[120], double x2[120],
                double y_1[120], double y_2[120],
                double A[120][120]) {


#pragma ACCEL PIPELINE flatten
for (int i = 0; i < 120; i++) {
  for (int j = 0; j < 120; j++) {
    x1[i] += A[i][j] * y_1[j];
  }
 }
#pragma ACCEL PARALLEL FACTOR=15
  for (int i = 0; i < 120; i++) {
#pragma ACCEL PARALLEL reduction = x2 FACTOR=12
    for (int j = 0; j < 120; j++) {
      x2[i] += A[j][i] * y_2[j];
    }
  }

}
```

Evaluation on Xilinx Vitis Library

- Tested on 33 kernels, each has 13.5 HLS optimization pragmas on average,
  - AutoDSE achieves roughly the same performance (1.04x higher)
  - Eliminated all HLS or Merlin optimization pragmas
- Both Merlin and AutoDSE keep and propagate dataflow and streaming pragmas
  - Will rely on dataflow composition using TAPA (later)



Current Goal: More Extensive DSE Using Deep Graph Learning

- Review of the problem

Manual code
- MVT kernel from Polybench
  - Two matrix-vetor multiplications

```
void kernel_mvt(double x1[120], double x2[120],
                double y_1[120], double y_2[120],
                double A[120][120]) {

for
  f
  }
}

for (int i = 0; i < 120; i++) {
  for (int j = 0; j < 120; j++) {
    x2[i] += A[j][i] * y_2[j];
  }
}

}
```

Solution space
- > 3M design choices

```
void kernel_mvt(double x1[120], double x2[120],
                double y_1[120], double y_2[120], double A[120][120]) {
#pragma ACCEL PIPELINE auto{__PIPE__L0}
#pragma ACCEL TILE FACTOR=auto{__TILE__L0}
                                                    RA__L2}
#pragma ACCEL PARALLEL reduction = x2 FACTOR=auto{__PARA__L3}
 } for (int j = 0; j < 120; j++) {
    x2[i] += A[j][i] * y_2[j];
}} } }
```

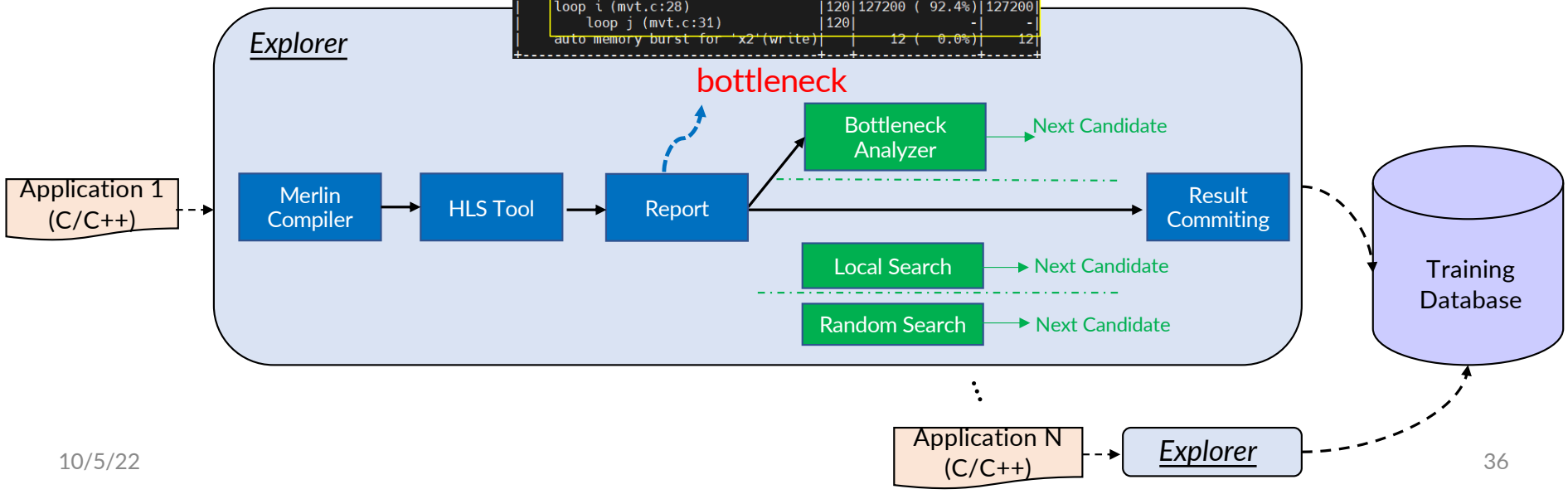
Solution:

Adopt a deep graph learning model to automatically learn the program's features

Step 1: Create a Database for Training the Model

- Database generation:
  - Adapting our previous work
    - AutoDSE [TODAES'22]

Hierarchy	TC	AC	CPC
kernel_mvt (mvt.c:4)		137701 (100.0%)	137701
auto memory burst for 'x1'(read)		120 (0.1%)	120
auto memory burst for 'A'(read)		7200 (5.2%)	7200
auto memory burst for 'y_1'(read)		120 (0.1%)	120
loop i (mvt.c:15)	120	727 (0.5%)	727
loop j (mvt.c:18)	120	-	-
auto memory burst for 'y_2'(read)		5 (0.0%)	5
auto memory burst for 'x1'(write)		120 (0.1%)	120
auto memory burst for 'A'(read)		246 (0.2%)	246
auto memory burst for 'x2'(read)		12 (0.0%)	12
loop i (mvt.c:28)	120	127200 (92.4%)	127200
loop j (mvt.c:31)	120	-	-
auto memory burst for 'x2'(write)		12 (0.0%)	12



Step 2: Represent the Program as a Graph

- Build the graph using the LLVM IR to capture lower-level instructions, i.e. closer to hardware
- Need to include both the program semantic and pragma flow in the graph
  - Program semantic: control, data, and call flow
    - Adapting the latest representation proposed for including these information (ProGraML [ICML'21])

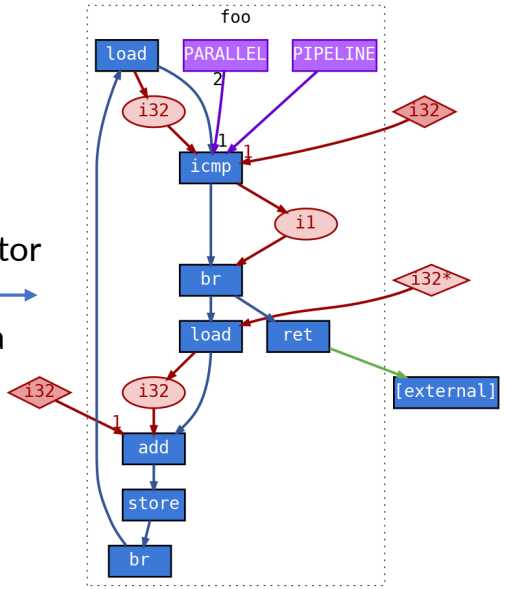
```
void foo(int input[N]) {

#pragma ACCEL PIPELINE auto{__PIPE__L1}
#pragma ACCEL PARALLEL FACTOR=auto{__PARA__L1}
  for (int i = 0; i < N; i++) {
    input[i] += 1;
  }
}
```

LLVM IR

```
. . .

for.cond:
  %0 = load i32, i32* %i, align 4
  %cmp = icmp slt i32 %0, 10
  br i1 %cmp, label %for.body, label %for.end

for.body:                                         %
1 = load i32, i32* %i, align 4
  %idxprom = sext i32 %1 to i64
  %arrayidx = getelementptr inbounds [10 x i32],
[10 x i32]* %a, i64 0, i64 %idxprom
  %2 = load i32, i32* %arrayidx, align 4
  %inc = add nsw i32 %2, 1
  store i32 %inc, i32* %arrayidx, align 4
  br label %for.inc

. . .
```

Graph generator

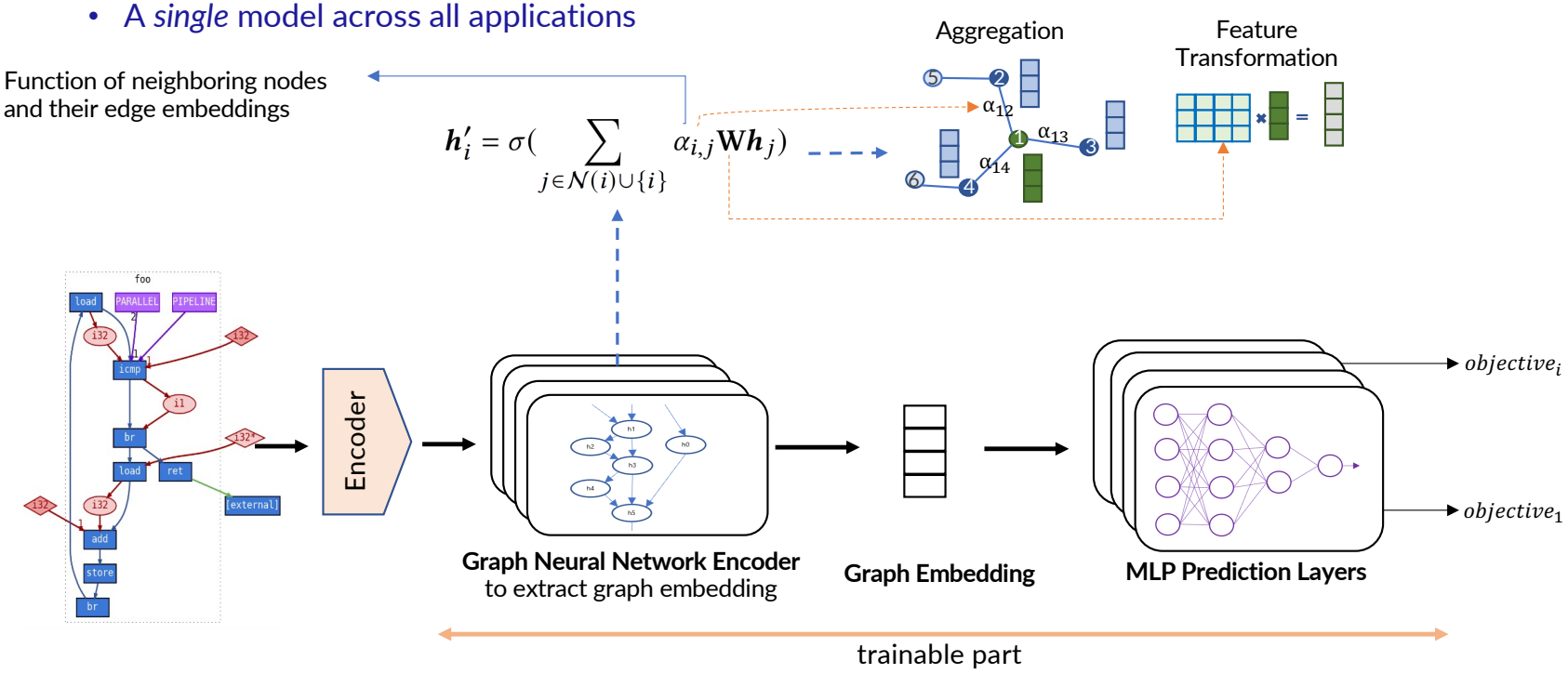
With pragma placeholder



- The graph is generated once per kernel and filled with different pragma values later on

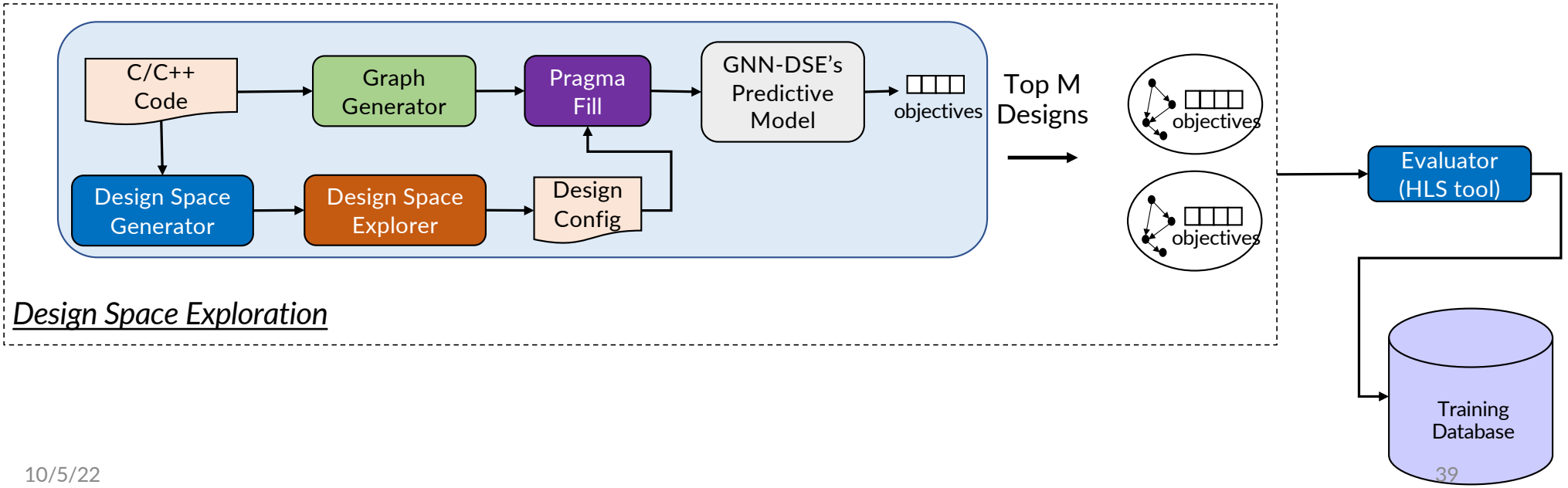
Step 3: Build a Predictive Model

- GNN-based model:
  - A *single* model across all applications



Design Space Exploration in GNN-DSE

- The trained model is replaced with the HLS tool for evaluating the design points
- The top M design points are evaluated with the HLS tool and added to the training database for subsequent trainings

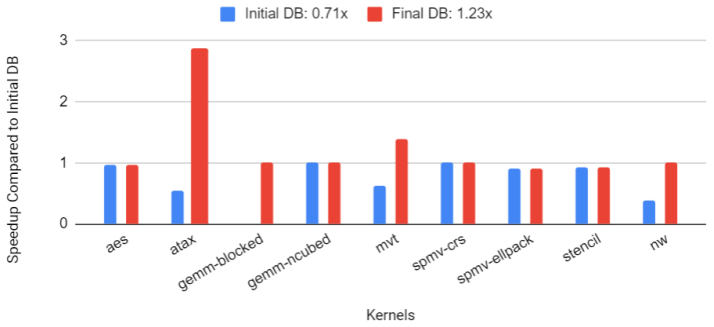


Experimental Results

- Model's performance
  - Regression loss is in RMSE

Model	Method	Speedup	DSP	LUT	FF	BRAM	All	Accuracy	F1-score
M1	MLP-pragma (based on Kown, et al. MLCAD'20)	3.28	0.59	0.31	0.25	0.34	4.76	0.52	0.42
M2	M1 + program context	2.94	0.47	0.24	0.13	0.16	3.94	0.78	0.40
M3	**GNN-DSE**	**0.56**	**0.13**	**0.08**	**0.06**	**0.05**	**0.85**	**0.93**	**0.87**

- Keep augmenting database until design space exploration (DSE) matches the best designs
  - Initial database:
    - 4428 total configs / 1036 valid configs
  - Final database:
    - 4752 total configs / 1278 valid configs
- More training examples lead to better accuracy



Experimental Results on Unseen Kernels

- DSE results on new kernels which were not in the database
  - All new kernels dealing with matrix vector operations
    - But with different coding styles, input sizes, and loop trip counts from our database
- Baseline: AutoDSE after 21 h
- GNN-DSE could achieve about the same performance
  - From −2% and +5% difference with a mean of +1%
  - With a maximum DSE time of 1 hour
- Adapting to domain shift in “Improving GNN-Based Accelerator Design Automation with Meta Learning [DAC’22]”

Kernel	# pragma	# Design configs	DSE + HLS Runtime (mins)	# Explored	Runtime Speedup
bicg	5	3,536	18	3,536	69x
doitgen	6	179	16	179	11x
gesummv	4	1,581	16	1,581	79x
2mm	14	492,787,501	74	78,676	17x

Current Limitation of GNN-DSE – Domain Shift

- Experimental evidence
  - Trained on a suite of 9 kernels
  - Tested on 5 different kernels with only 20 labeled designs for each of the 5 new kernel
  - Root mean square error (RMSE) on the hold-out test set of each new kernel

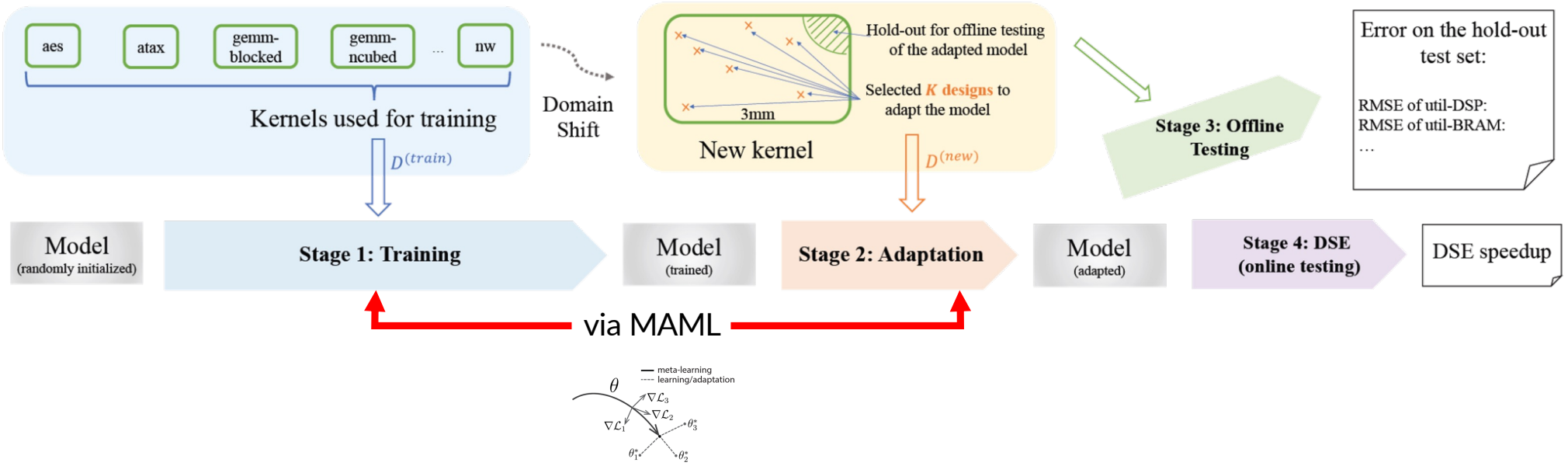
	jacobi-1d	fdtd-2d	gemm	3mm	gemver
GNN-DSE	4.2496	6.7047	7.5337	9.1584	4.4717

  - DSE speedup with respect to AutoDSE after 20 hours

	jacobi-1d	fdtd-2d	gemm	3mm	gemver
GNN-DSE	0.44x	0.06x	0.87x	0.30x	0.20x

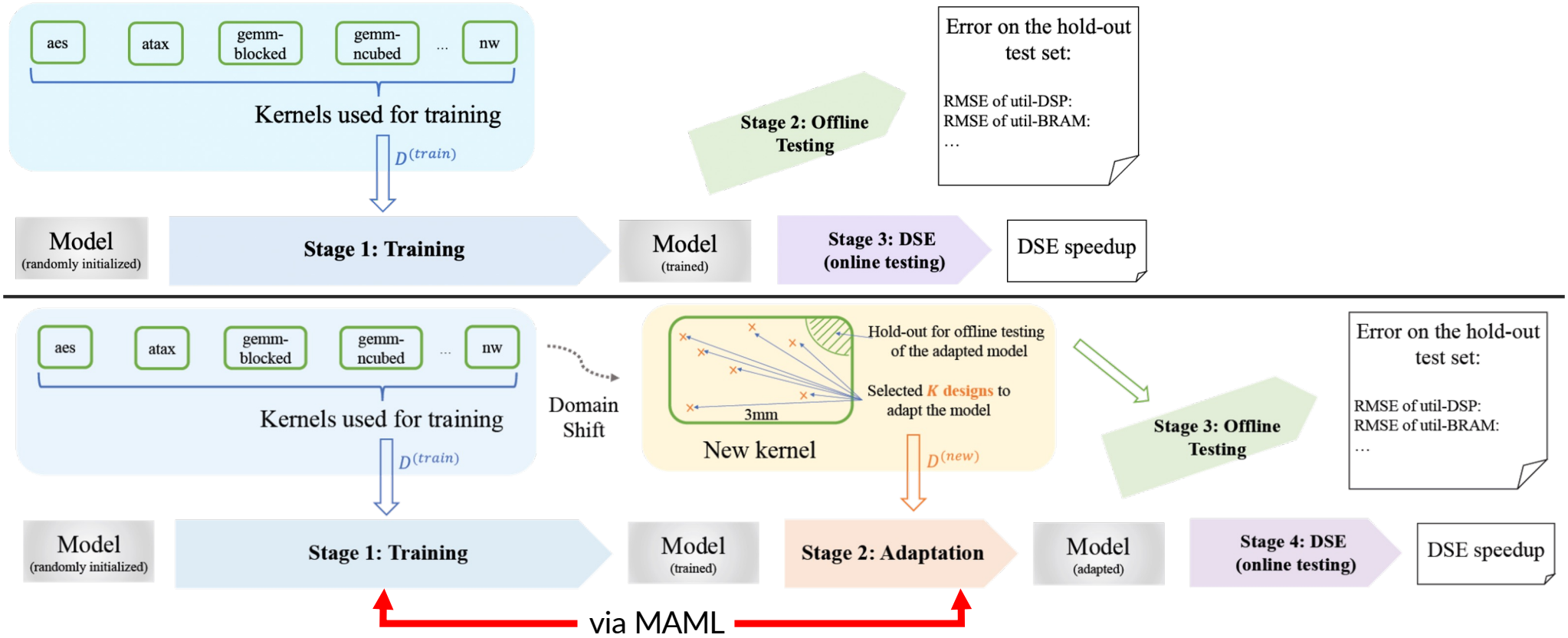
  - Accuracy drops when the testing kernels differ a lot from the training ones (domain shift), causing unsatisfactory DSE results. Meanwhile, our goal is to design a method that works well on any real-world kernel.

Proposal: Use Transfer Learning (GNN-DSE-MAML)



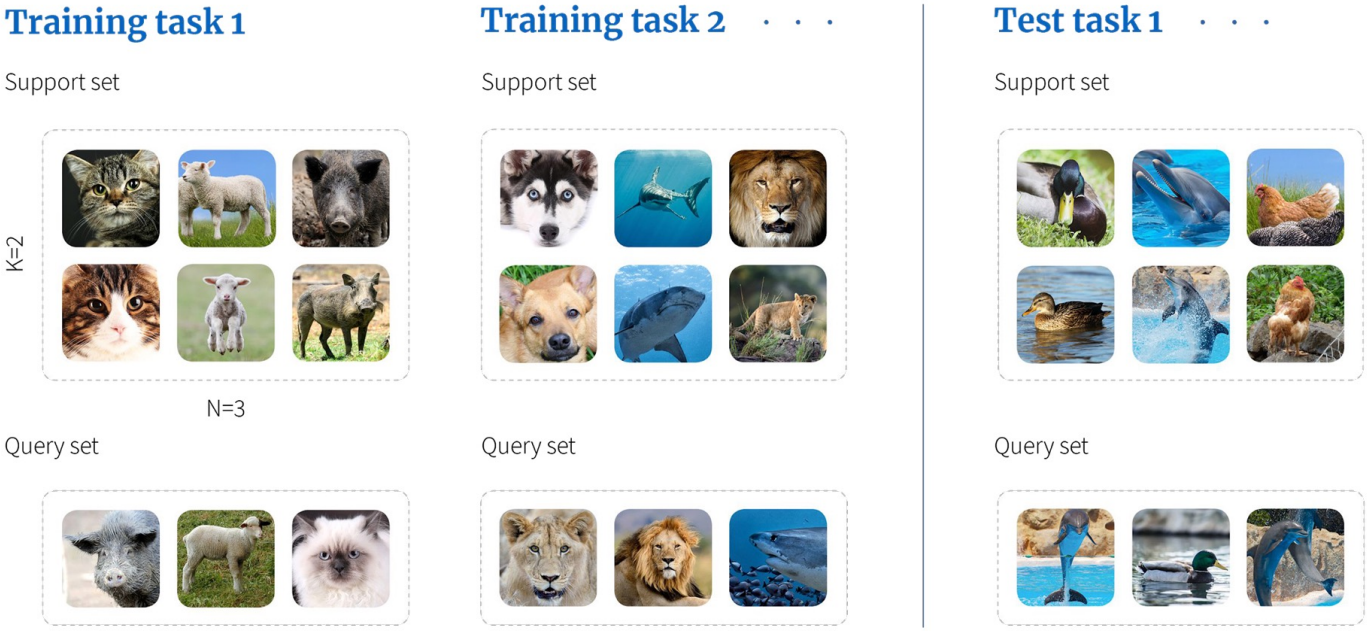
Model-Agnostic Meta-Learning (MAML) -- *Finn et al. 2017.*

GNN-DSE (top) vs GNN-DSE-MAML (bottom)



Inspiration: K-shot Image Classification Using Meta-Learning

- **Meta-learning:**
  - Compute a model that can eventually generalize across many tasks
  - with good data and computation efficiency:
- **Example:**
  - K-shot image classification task:
  - learn a classification model that can quickly adapt to a new class with only K images from that class



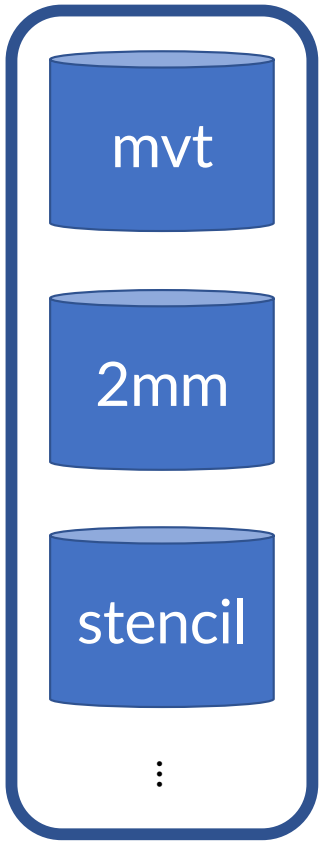
MAML for Training

Algorithm 1 Training procedure of GNN-DSE-MAML

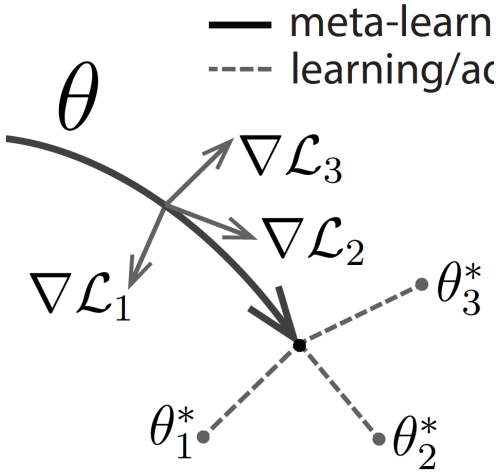
Require: $p(\mathcal{P}^{(train)})$: distribution over kernels (programs) for training

Require: α, β: step size hyperparameters

1: randomly initialize θ
2: **while** not done **do**
3: Sample batch of kernels $\mathcal{P}_i \sim p(\mathcal{P}^{(train)})$
4: **for all** $\mathcal{P}_i$ **do**
5: Sample K datapoints $\mathcal{D} = \{X_j, Y_j\}$ from $\mathcal{P}_i$
6: Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{P}_i}(f_\theta)$ using $\mathcal{D}$ and $\mathcal{L}_{\mathcal{P}_i}$ in Equation 1
7: Compute adapted parameters with gradient descent: $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{P}_i}(f_\theta)$
8: Sample datapoints $\mathcal{D}_i' = \{\mathbf{x}^{(j)}, \mathbf{y}^{(j)}\}$ from $\mathcal{P}_i$ for the meta-update
9: **end for**
10: Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{P}_i \sim p(\mathcal{P})} \mathcal{L}_{\mathcal{P}_i}(f_{\theta_i'})$ using each $\mathcal{D}_i'$ and $\mathcal{L}_{\mathcal{P}_i}$ in Equation 1
11: **end while**



A batch of kernels



MAML for Adaptation

Algorithm 1 Training procedure of GNN-DSE-MAML

Require: $p(\mathcal{P}^{(train)})$: distribution over kernels (programs) for training

Require: α, β: step size hyperparameters

1: randomly initialize θ

2: **while** not done **do**

3: Sample batch of kernels $\mathcal{P}_i \sim p(\mathcal{P}^{(train)})$

4: **for all** $\mathcal{P}_i$ **do**

5: Sample K datapoints $\mathcal{D} = \{X_j, Y_j\}$ from $\mathcal{P}_i$

6: Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{P}_i}(f_\theta)$ using $\mathcal{D}$ and $\mathcal{L}_{\mathcal{P}_i}$ in Equation 1

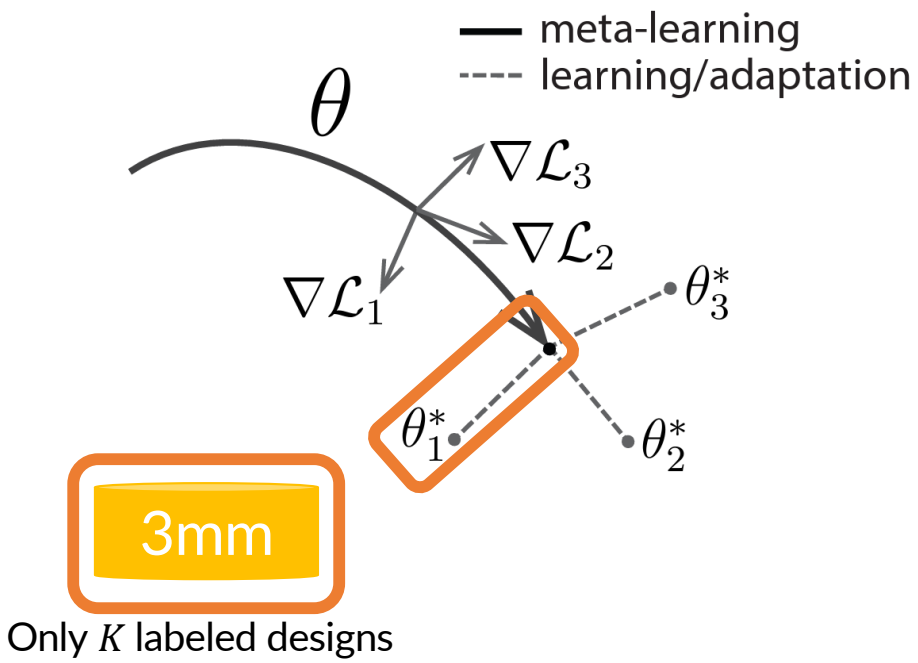
7: Compute adapted parameters with gradient descent: $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{P}_i}(f_\theta)$

8: Sample datapoints $\mathcal{D}_i' = \{\mathbf{x}^{(j)}, \mathbf{y}^{(j)}\}$ from $\mathcal{P}_i$ for the meta-update

9: **end for**

10: Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{P}_i \sim p(\mathcal{P})} \mathcal{L}_{\mathcal{P}_i}(f_{\theta_i'})$ using each $\mathcal{D}_i'$ and $\mathcal{L}_{\mathcal{P}_i}$ in Equation 1

11: **end while**



Only K labeled designs

Experimental Results – Offline Testing

- K=20 for adaption
- Adaptation is necessary for the unadapted model to obtain lower error
- FineTune: Naïve adaptation using the regular objective function
- Under 4 out of 5 kernels, MAML leads to a more accurate adapted model.

Method	jacobi-1d	fdtd-2d	gemm	3mm	gemver
GNN-DSE-UNADAPTED	4.2496	6.7047	7.5337	9.1584	4.4717
GNN-DSE-FINETUNE	3.2611	4.0831	**1.7342**	6.2930	3.1600
GNN-DSE-MAML	**2.3898**	**2.4912**	2.1116	**5.9670**	**3.0303**

Experimental Results – DSE

- MAML-based adaptation achieves great performance for 3 new kernels
  - 3mm: >17 trillion design candidates that AutoDSE got to explore only 149 of them after 20 hours since it relies on the HLS tool for evaluating each candidate
    - GNN-DSE-MAML yields a significant speedup for 3mm compared to AutoDSE

Method	jacobi-1d	fdtd-2d	gemm	3mm	gemver
GNN-DSE-UNADAPTED	0.44×	0.06×	0.87×	0.30×	0.20×
GNN-DSE-FINETUNE	0.54×	0.04×	0.18×	1.00×	0.22×
GNN-DSE-MAML	1.00×	TO	1.21×	64.52×	TO

TO: Timed Out

Experimental Results – DSE

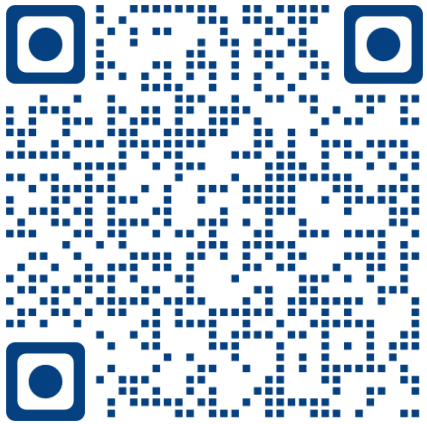
- For ftdt-2d and gemver, the MAML results lead to Timed Out
  - The MAML-based model uses high degree of parallelization for each section of the loop nests, overwhelming the HLS tool.
  - Such cases were not covered in the K sampled samples for adapting the model.

Method	jacobi-1d	fdtd-2d	gemm	3mm	gemver
GNN-DSE-UNADAPTED	0.44×	0.06×	0.87×	0.30×	0.20×
GNN-DSE-FINETUNE	0.54×	0.04×	0.18×	1.00×	0.22×
GNN-DSE-MAML	1.00×	TO	1.21×	64.52×	TO

TO: Timed Out

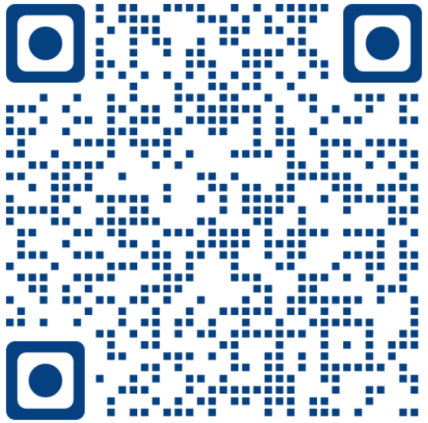
AutoDSE and GNN-DSE are Open-source

- https://github.com/UCLA-VAST/GNN-DSE



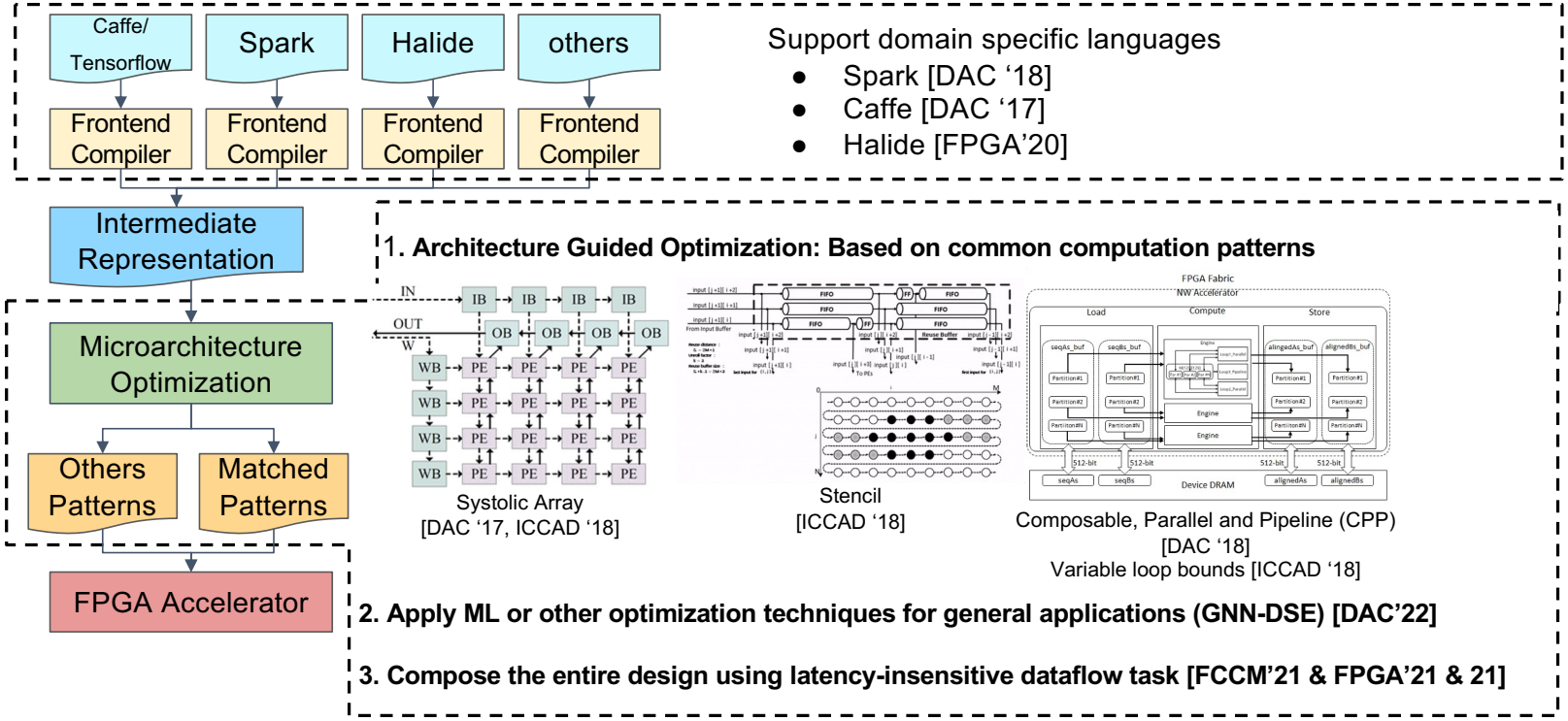
Scan me!

- https://github.com/UCLA-VAST/AutoDSE



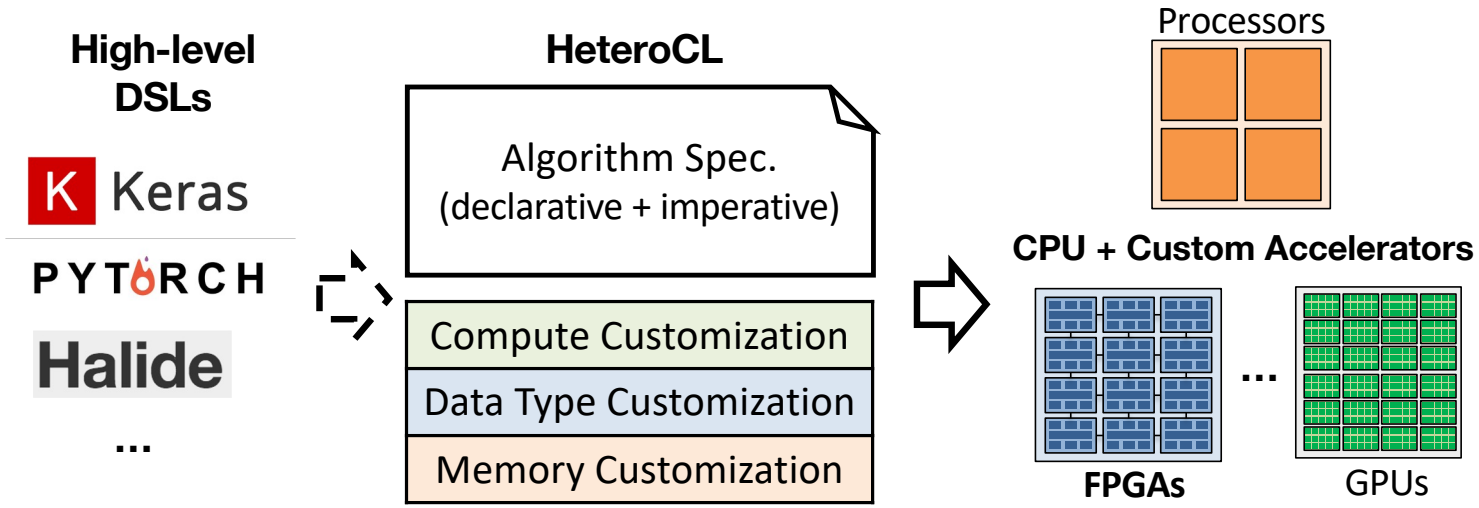
Scan me!

How to Integrate Different Approaches?



HeteroCL Programming Infrastructure [FPGA'19]

- **Inspired by Halide: Separate program specification and optimization (scheduling)**
  - **Flexible: Mixed declarative & imperative programming**
  - **Portable: Clean decoupling of algorithm & hardware customizations**
  - **Efficient: Mapping to high-performance spatial architecture templates**



Open-source: https://github.com/cornell-zhang/heterocl

HeteroCL in a Nutshell

HeteroCL code

Algorithm

```
r = hcl.reduce_axis(0, 3)
c = hcl.reduce_axis(0, 3)
out = hcl.compute(N, N),
    lambda y, x:
        hcl.sum(image[x+r, y+c]*kernel[r, c],
            axis=[r, c]))
```

Declarative code (based on TVM)

Custom Compute

```
s = hcl.create_schedule()
s[out].unroll([r,c])
```

Custom Data Type

```
for i in range(2, 8):
  s.quantize([out], Fixed(i, i-2))
```

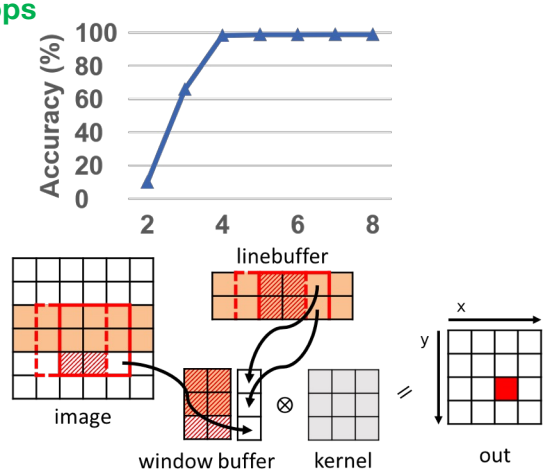
Custom Memory

```
linebuf = s[image].reuse_at(out, out.y)
winbuf = s[linebuf].reuse_at(out, out.x)
```

Corresponding C code

```
for (int y = 0; y < N; y++)
  for (int x = 0; x < N; x++)
    for (int r = 0; r < 3; r++)
      for (int c = 0; c < 3; c++)
        out[x, y] += image[x+r, y+c] * kernel[r, c]
```

Unroll inner loops

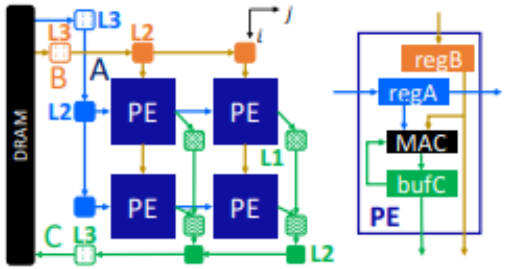


Y.-H. Lai, et al., *HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing*, **FPGA'2019 Best Paper Award**

HeteroCL: Mapping to Spatial Architecture Templates

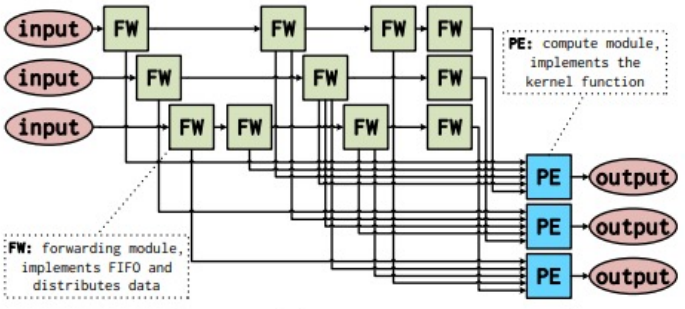
- **Systolic Array**

```
# matrix multiply kernel
out = hcl.compute(N, N),
    lambda y, x: sum(A[x, k] * B[k, y]), axis=k)

s[out].systolic()
```



- **Stencil Architecture**

```
# jacobi kernel
out = hcl.compute(N, N),
    lambda y, x:
        (in[y,x-1]+ in[y-1,x] + in[y,x] + in[y,x+1] + in[y+1,x])/5))

s[out].stencil()
```

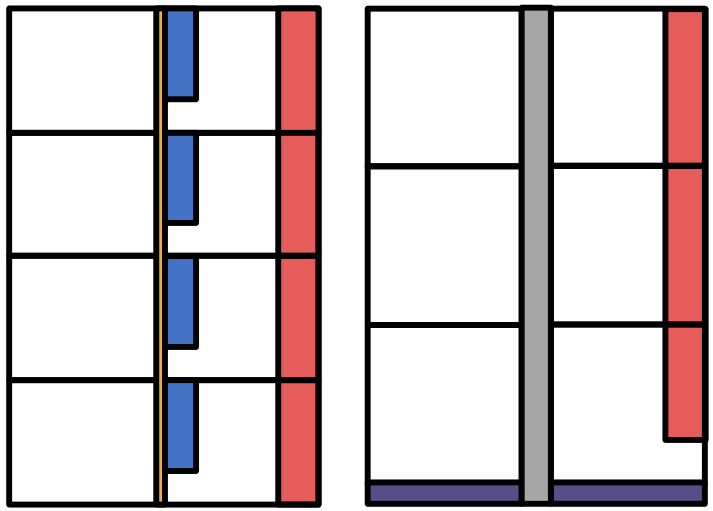


One More Question:

Now I am good at using (enhanced) HLS, how to deal with (low) clock frequency and (long) compilation time from downstream physical synthesis ?

Modern FPGAs are Large and Complex

- FPGAs are increasingly large
- Multiple dies integrated together
- High delay penalty for die-crossing
- Large IPs with pre-determined location

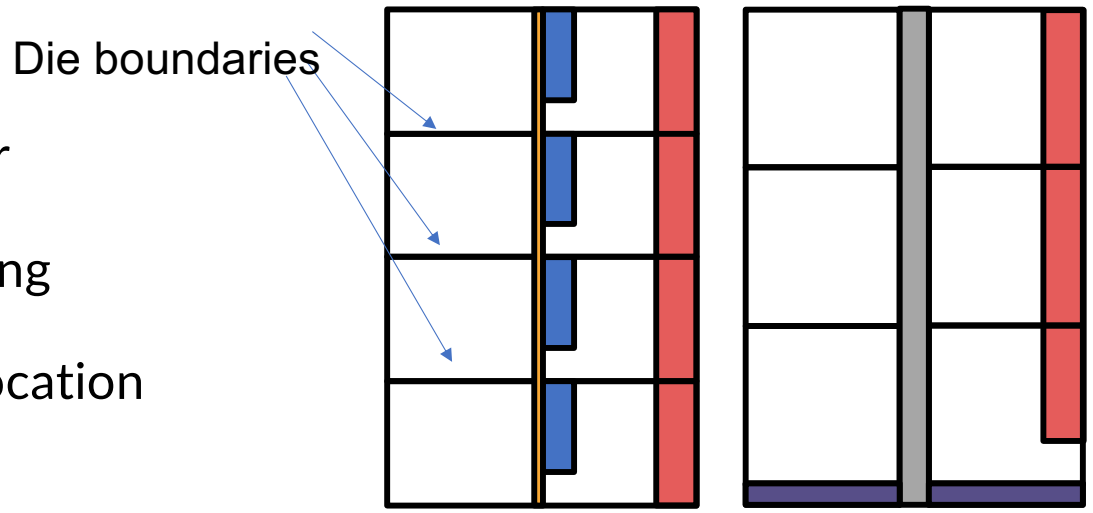


Xilinx Alveo U250

Xilinx Alveo U280

Modern FPGAs are Large and Complex

- FPGAs are increasingly large
- Multiple dies integrated together
- High delay penalty for die-crossing
- Large IPs with pre-determined location

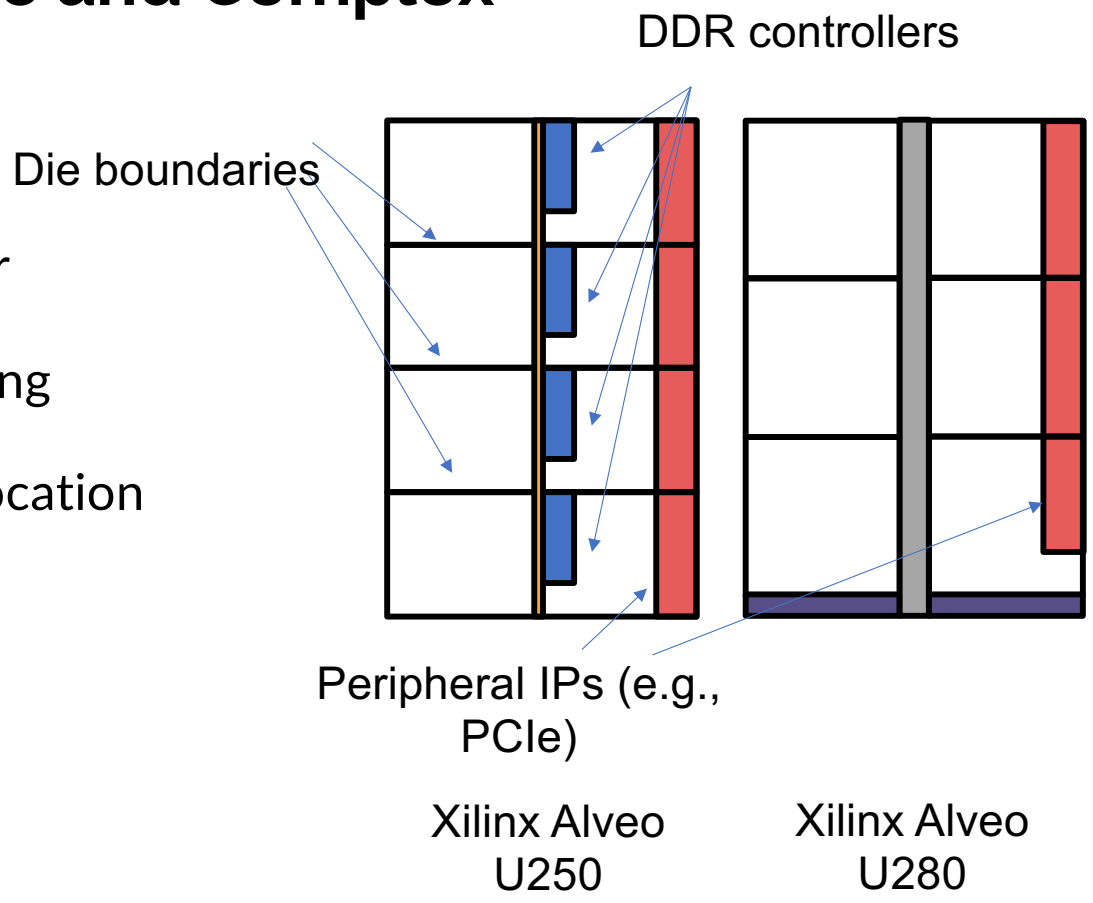


Xilinx Alveo U250

Xilinx Alveo U280

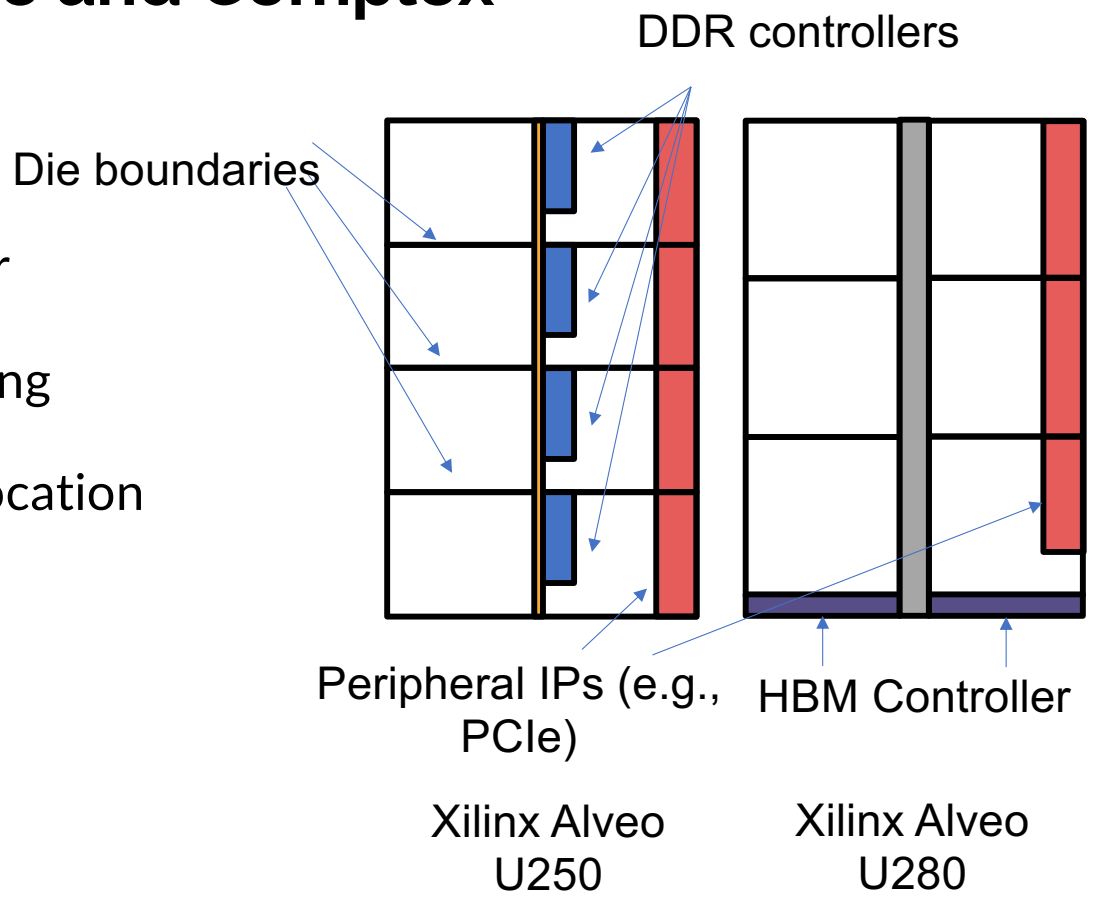
Modern FPGAs are Large and Complex

- FPGAs are increasingly large
- Multiple dies integrated together
- High delay penalty for die-crossing
- Large IPs with pre-determined location



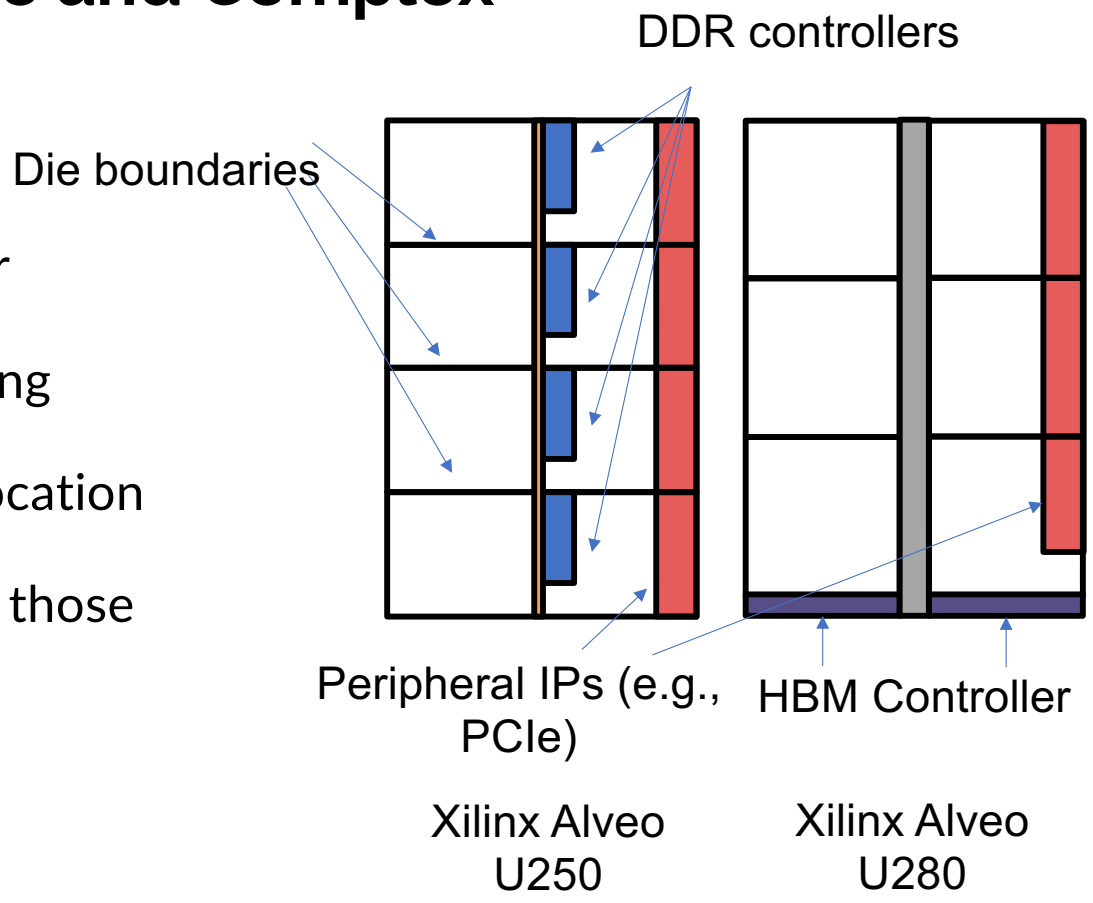
Modern FPGAs are Large and Complex

- FPGAs are increasingly large
- Multiple dies integrated together
- High delay penalty for die-crossing
- Large IPs with pre-determined location



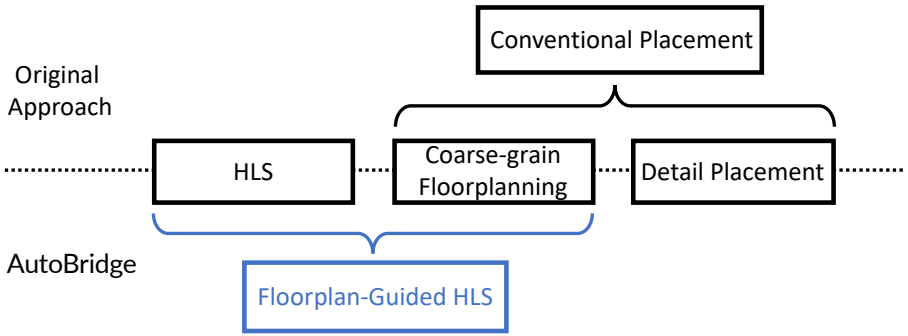
Modern FPGAs are Large and Complex

- FPGAs are increasingly large
- Multiple dies integrated together
- High delay penalty for die-crossing
- Large IPs with pre-determined location
- HLS has limited consideration of those physical barriers



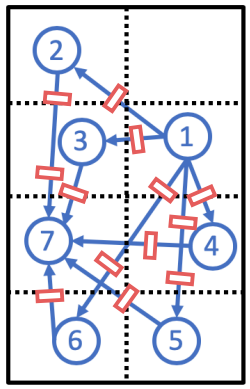
AutoBridge [FPGA'21 Best Paper Award]

- Add extra pipeline stages to long interconnects
- Couples floorplanning with HLS pipelining
- Global optimization to assure correctness
- Automate latency-insensitive design at the HLS level
- Improve average frequency from 150 MHz to 297 MHz over 43 test cases.



Successful Applications:

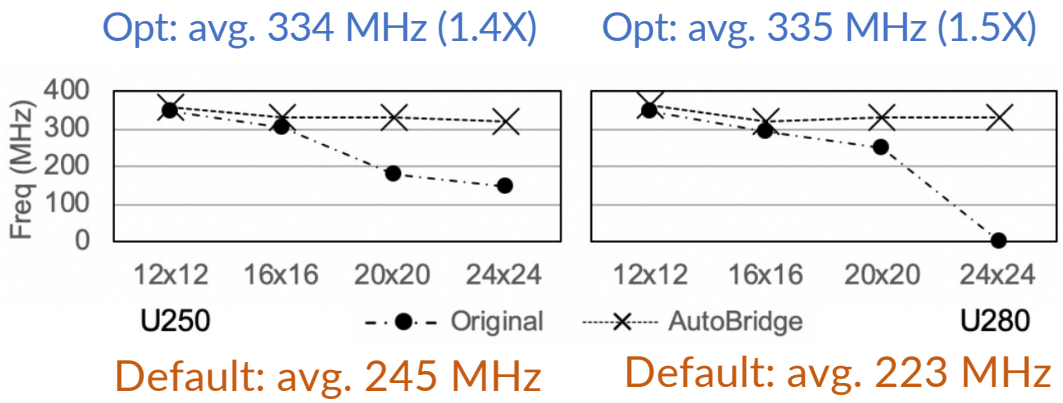
- [FPGA'21] AutoSA: A polyhedral compiler for high-performance systolic arrays on fpga
- [FPGA'22] Accelerating SSSP for Power-Law Graphs
- [FPGA'22] Sextans: A Streaming Accelerator for General-Purpose Sparse-Matrix Dense-Matrix Multiplication
- [DAC'22] Serpens: A High Bandwidth Memory Based Accelerator for General-Purpose Sparse Matrix-Vector Multiplication
- .....



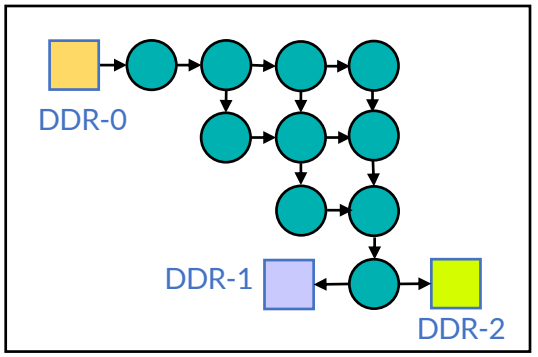
Insert pipeline registers after floorplanning to fix critical paths

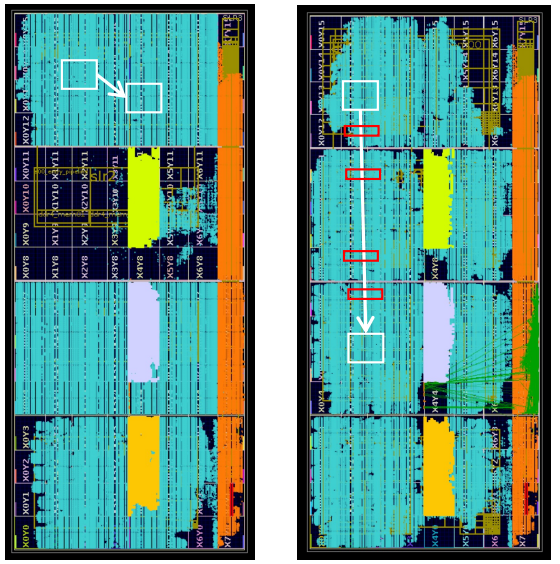
Case Study

- Gaussian Elimination, 8 configurations



- Difference in Resource Utilization
  - LUT: -0.14%
  - FF: -0.04%
  - BRAM: -0.03%
  - DSP: +0.00%

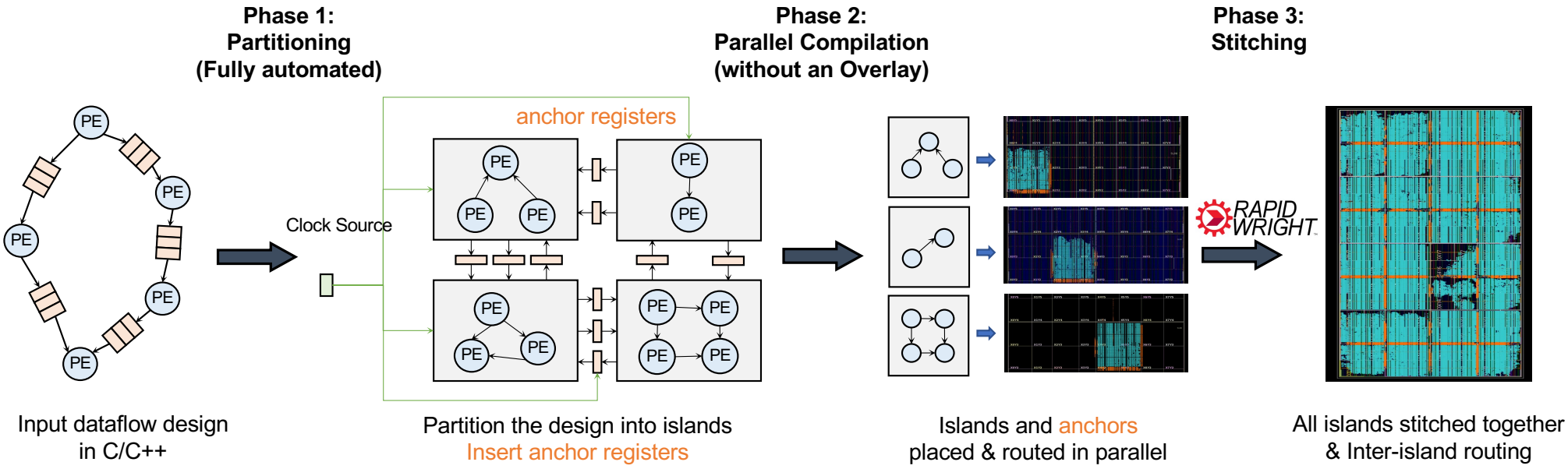




Default AutoBridge

Comparison of the 24x24 Design on U250

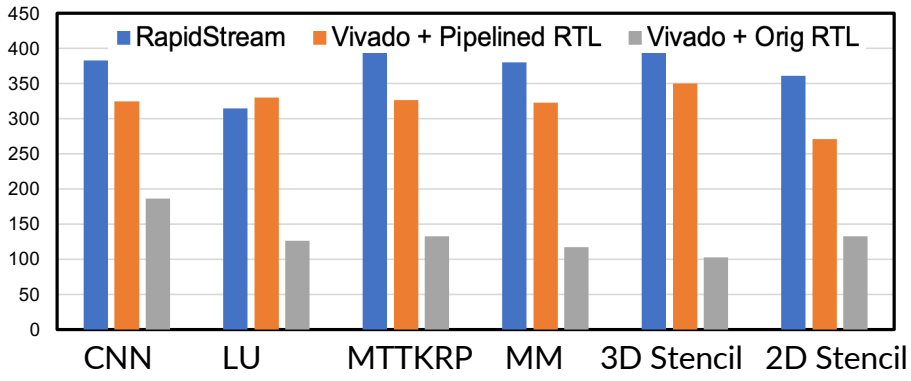
Latency-Insensitive Designs Helped Compile Time as Well!



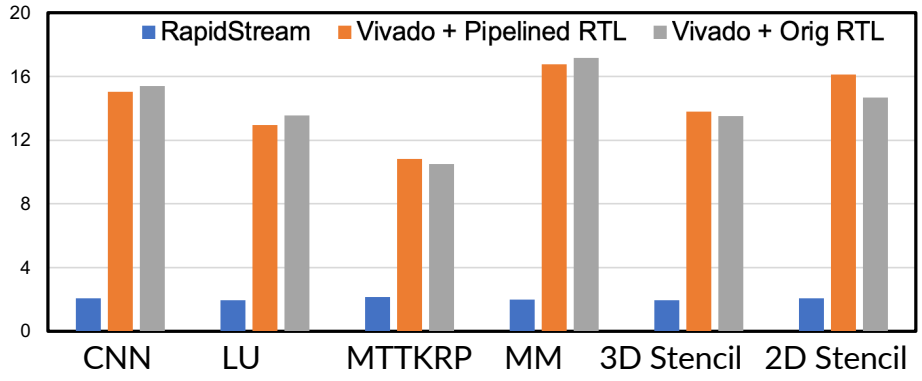
RapidStream [FPGA '22 Best Paper Award]

Experimental Result

- Tested on 6 large scale dataflow designs targeting Xilinx U250 FPGA with 4 SLRs (dies)
- Distribute to 4 Xeon servers, each with 56 cores
- Divide the FPGA into 32 islands (8 rows, 4 columns)
- 5-7X speedup (from C++ to fully routed checkpoint)
- Up to 1.3X frequency improvement

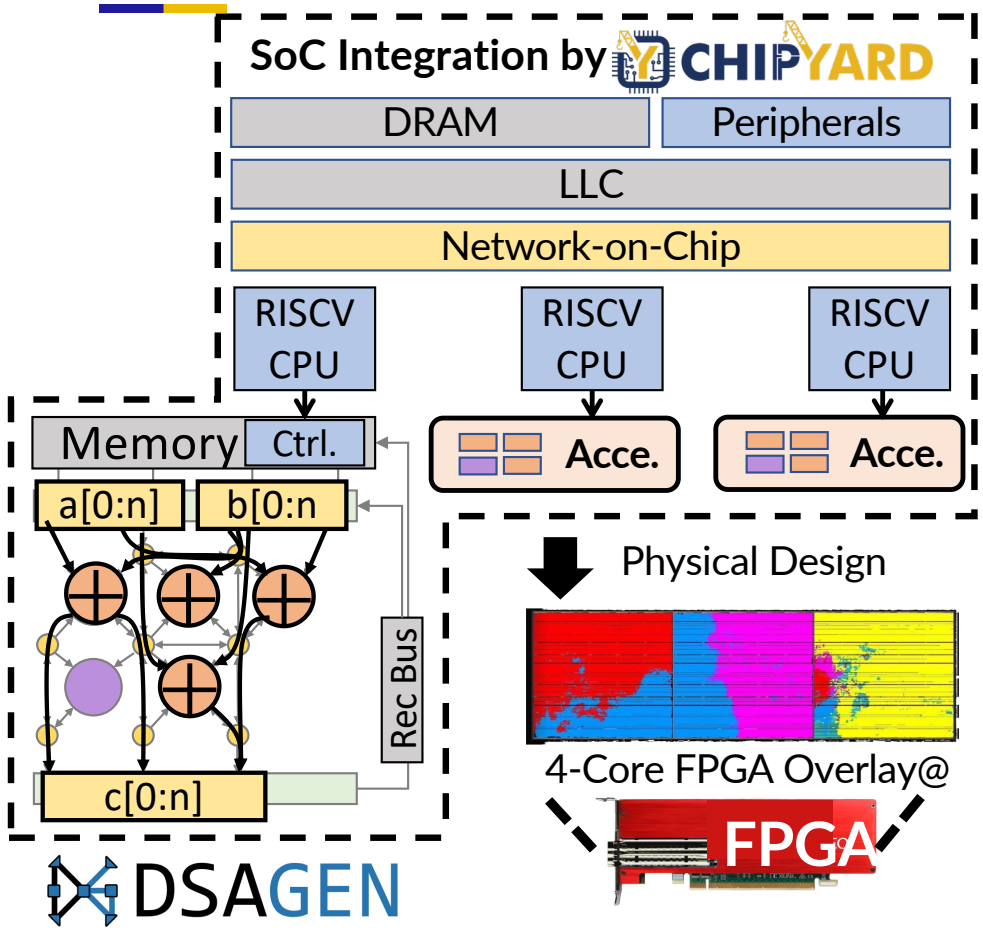


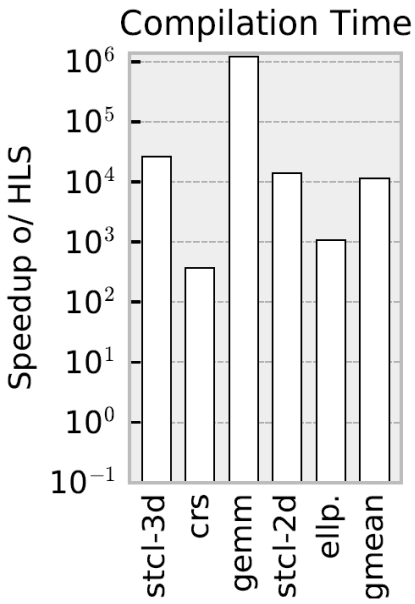
Frequency (MHz)



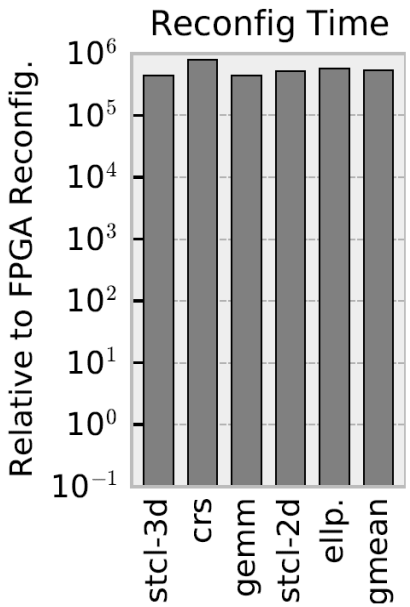
Runtime (hour)

Use Overlay for Even Faster Compilation: OverGen [MICRO'22]





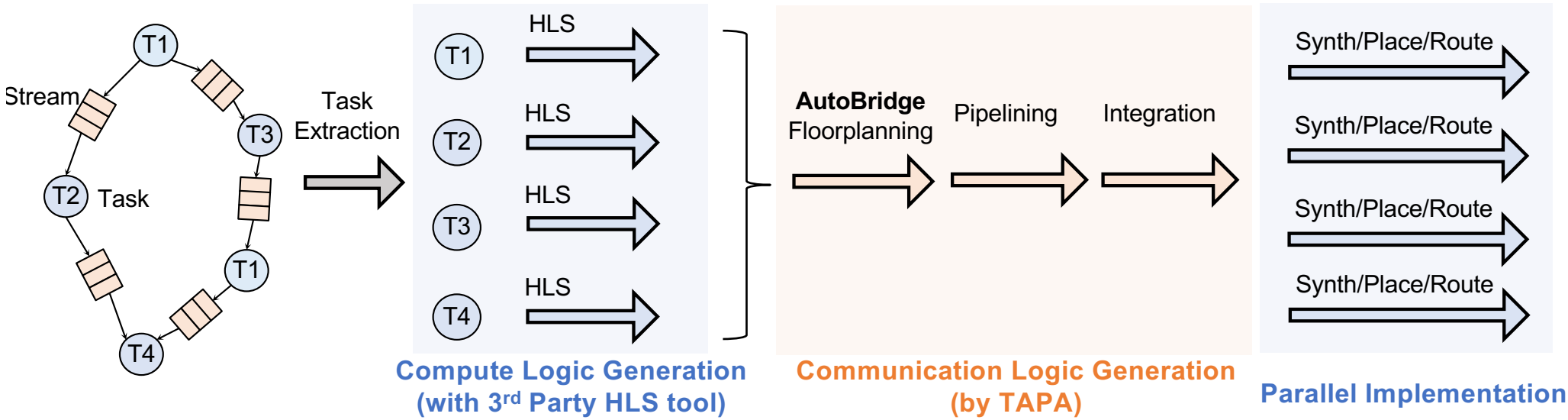
10,000x faster in re-compile



100,000x faster in reconf.

Composing Large Dataflow Designs Using TAPA

- TAPA programs explicitly decouple communication and computation
- Computation => compiled by Vitis HLS / AutoSA / AutoDSE / …
- Communication => generated by TAPA

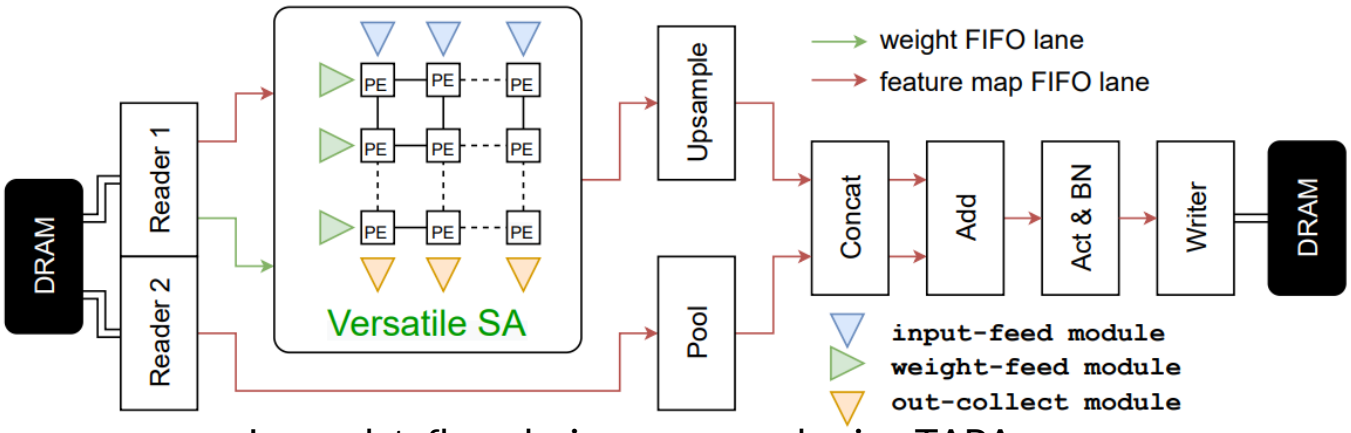


Example: FlexCNN Using TAPA

- **FlexCNN: an end-to-end automated DNN synthesis framework**
- **From ONNX to bitsream on FPGAs**

Module	Lines of Code	Code Generation
Reader 1	1,046	Template-based
Reader 2	446	Template-based
Systolic Array	4,801	Automatic
Pool	254	Template-based
Upsample	221	Template-based
Concat	350	Template-based
Add	314	Template-based
Act & BN	320	Template-based
Writer	824	Template-based
Top	6,292	Automatic
Total	**14,868**	

FlexCNN without TAPA	FlexCNN with TAPA
Fails Placement & Route	Achieves up to 266 MHz



Large dataflow design composed using TAPA

Concluding Remark 1

- I am encouraged by the progress/results on democratizing accelerator designs and customized computing
- It takes a community-wide effort
- All our tools are open-sourced, and FPGA vendors are more open as well
  - One-API from Intel
  - Merlin from AMD/Xilinx (after acquisition of Falcon Computing)
- Increasingly interested in using MLIR as an integration point

Concluding Remark 2

- **Important for the architecture community to have a rapid prototyping flow**
  - **From Idea to Silicon in days, not months/years**
- **Concerned with some accelerator evaluation methodology**
  - ***"We evaluate XXX using a C++-based cycle-level simulator."***
  - ***Does it consider***
    - reduced memory bandwidth due to short burst length?
    - interconnect network size and latency from HBM ports to logic elements?
    - interconnect delays …?
  - **Has it been validated against any real silicon (FPGA or ASIC)?**

Concluding Remark 3

- **I had the pleasure working with many collaborators in other application domains.**


Alex Bui and William Hsu
Low-dose CT reconstruction

Tad Blair
Real-time neural signal processing

Prof. Yizhou Sun
(UCLA)

Yizhou Sun
Graph similarity computation

- **It's time to enable domain experts to design their own accelerators!**
- **The deep learning community has done a much better job – "every" domain expert can train complex DL models**
- **Can we catch up? Think about broader impact!**

Final Remark

- No doubt we are in an exciting era for computer architecture
- We want to every (serious) software programmer to participate
  - Not just architects
- Build his/her own customized accelerators on field-programmable fabrics
  - On premise or in the cloud
- I hope that many of you can join this effort

2018 Turing Award Lecture

A Story …

- **Q: Does everyone here do High-Level Synthesis?**
- **A: What do you mean? We are all from Harvard Law School.**

Acknowledgements:
NSF, JUMP/CRISP, and CDSC Industrial Partners

- Multi-year efforts by many students, postdocs, and collaborators


Prof. Tony Nowatzki (UCLA)

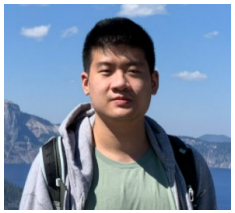

Prof. Yizhou Sun (UCLA)

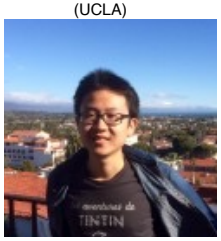

Prof. Miryung Kim (UCLA)


Prof. Zhiru Zhang (Cornell Univ.)


Prof. Peipei Zhou (Univ. of Pittsburgh)


Prof. Vivek Sarkar (Georgia Tech)

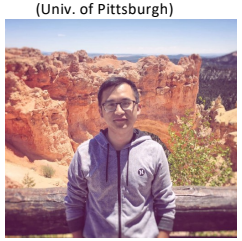

Yunsheng Bai (UCLA)

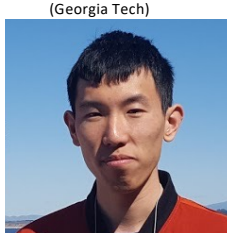

Jie Wang (UCLA)


Peng Wei (UCLA)


Hao Yu (UCLA/Falcon)


Yi-Hsiang Lai (Cornell)

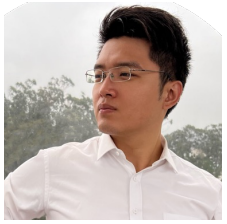

Weikang Qiao (UCLA)


Yuan Zhou (Cornell)

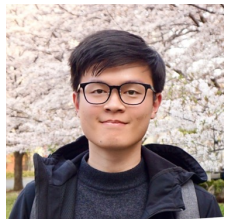

Zhengrong Wang (UCLA)


Yuze Chi (UCLA)


Atefeh Sohrabizadeh (UCLA)

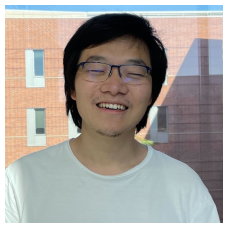

Sihao Liu (UCLA)


Zhe Chen (UCLA)


Jason Lau (UCLA)


Suhail Basalama (UCLA)


Licheng Guo (UCLA)


Jian Weng (UCLA)

Thank You!