UCLA COMPUTER SCIENCE DEPARTMENT

Center for Domain Specific Computing

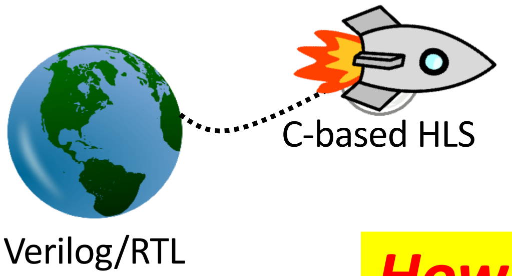
Democratize Customizable Computing

Jason Cong
Computer Science Department, UCLA

June 2019

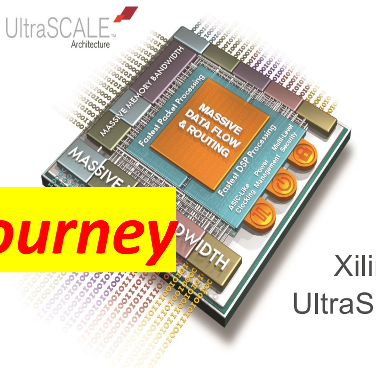
High-level synthesis for FPGA Programming is Real

- Here is the moon!



Better programmability
→ Faster prototyping
→ Faster development cycle

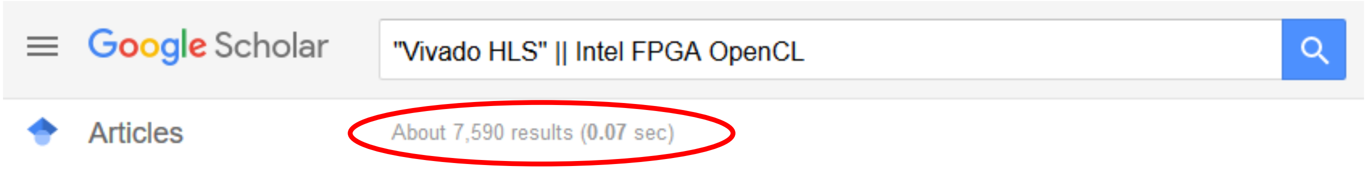
However, it's not an easy journey



Xilinx UltraScale+

Intel Stratix 10

- Commercial HLS tools are now widely used
  - xPilot (UCLA 2006) → AutoPilot (AutoESL) → Vivado HLS (Xilinx 2011-)
  - Intel® FPGA SDK for OpenCL™ (2016-)



"Vivado HLS" || Intel FPGA OpenCL

About 7,590 results (0.07 sec)

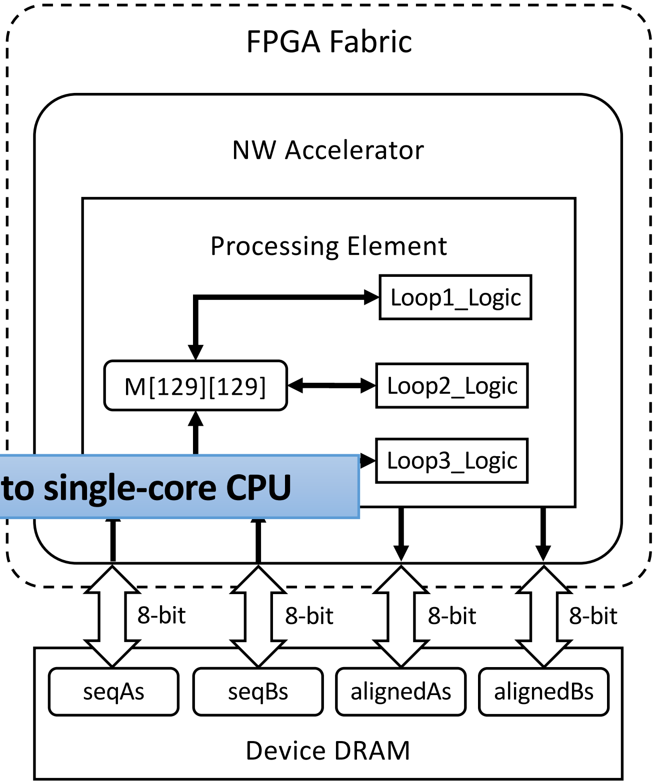
HLS Challenges and Solutions

- Challenge 1: Heavy code reconstruction
  - Modern HLS tools require particular coding style for performance

Not All C Programs Lead to Good Performance

- Example: The Needleman-Wunsch algorithm for sequence alignment

```
void engine(...) {
  int M[129][129];
  ...
loop1: for(i=0; i<129; i++) {M[0][i]=...}
loop2: for(j=0; j<129; j++) {M[j][0]=...}
loop3: for(i=1; i<129; i++) {
    for(j=1; j<129; j++) {...
      M[i][j]=...
  }}
  ...
}
void kernel(char seqAs[], char seqBs[],
       char alignedAs[], char alignedBs[]) {
  for (int i=0; i<NUM_PAIRS; i++) {
    engine(seqAs+i*128, seqBs+i*128,
       alignedAs+i*256, alignedBs+i*256);
}}
```



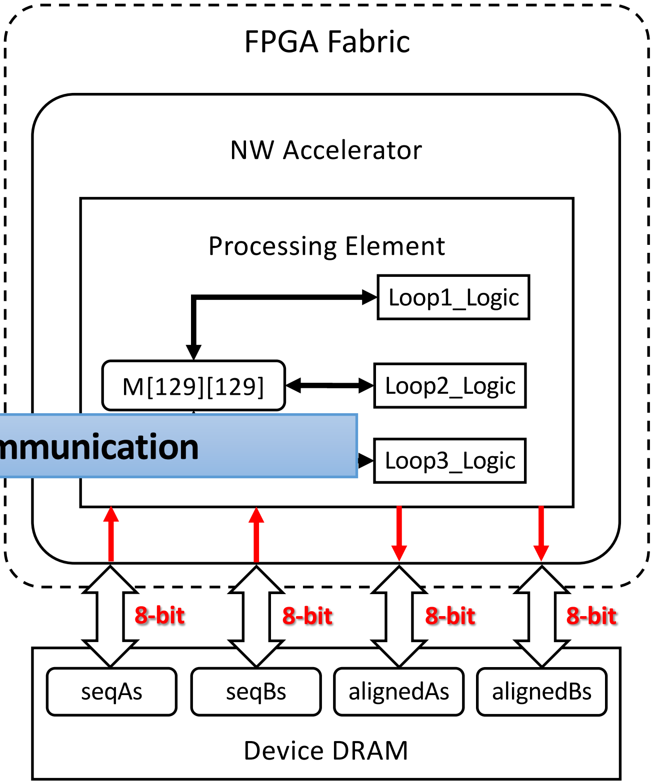
~100x slow down compared to single-core CPU

Not All C Programs Lead to Good Performance

- Example: The Needleman-Wunsch algorithm for sequence alignment

```
void engine(...) {
  int M[129][129];
  ...
loop1: for(i=0; i<129; i++) {M[0][i]=...}
loop2: for(j=0; j<129; j++) {M[j][0]=...}
loop3: for(i=1; i<129; i++) {
    for(j=1; j<129; j++) {...
      M[i][j]=...
  }}
  ...
}
void kernel(char seqAs[], char seqBs[],
       char alignedAs[], char alignedBs[]) {
  for (int i=0; i<NUM_PAIRS; i++) {
    engine(seqAs+i*128, seqBs+i*128,
       alignedAs+i*256, alignedBs+i*256);
}}
```

Inefficient Off-Chip Communication

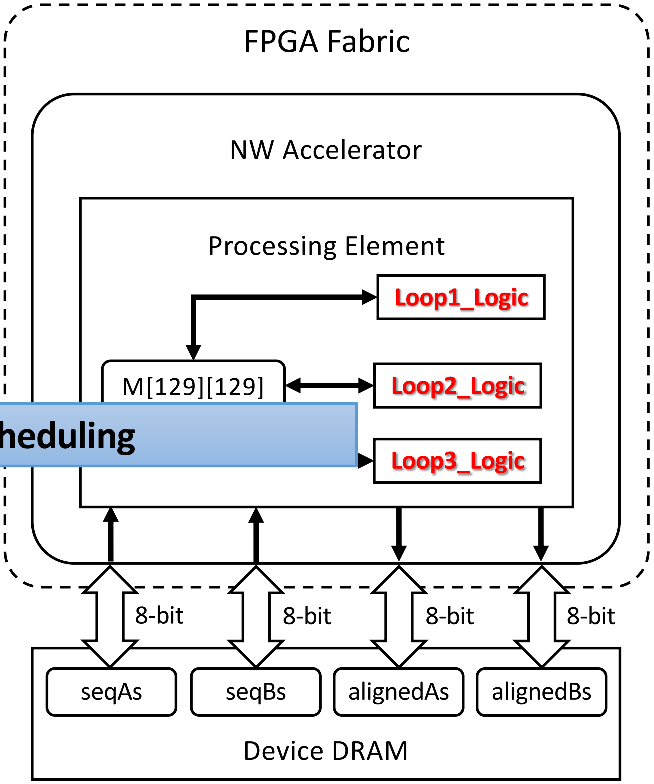


Not All C Programs Lead to Good Performance

◆ Example: The Needleman-Wunsch algorithm for sequence alignment

```
void engine(...) {
  int M[129][129];
  ...
loop1: for(i=0; i<129; i++) {M[0][i]=...}
loop2: for(j=0; j<129; j++) {M[j][0]=...}
loop3: for(i=1; i<129; i++) {
    for(j=1; j<129; j++) {...
      M[i][j]=...
  }}
  ...
}
void kernel(char seqAs[], char seqBs[],
       char alignedAs[], char alignedBs[]) {
  for (int i=0; i<NUM_PAIRS; i++) {
    engine(seqAs+i*128, seqBs+i*128,
       alignedAs+i*256, alignedBs+i*256);
}}
```

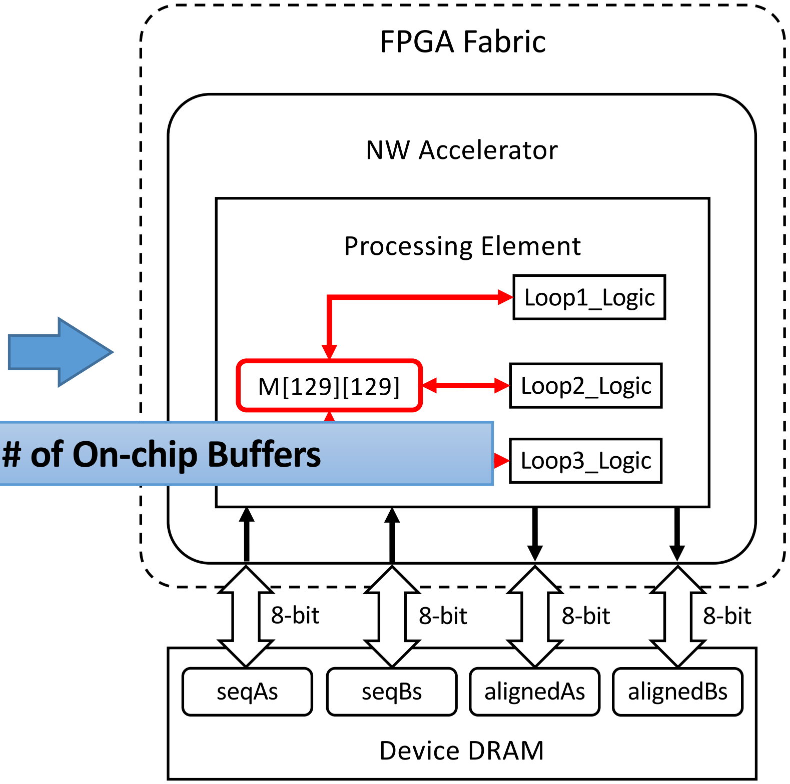
Sequential Loop Scheduling



Not All C Programs Lead to Good Performance

- Example: The Needleman-Wunsch algorithm for sequence alignment

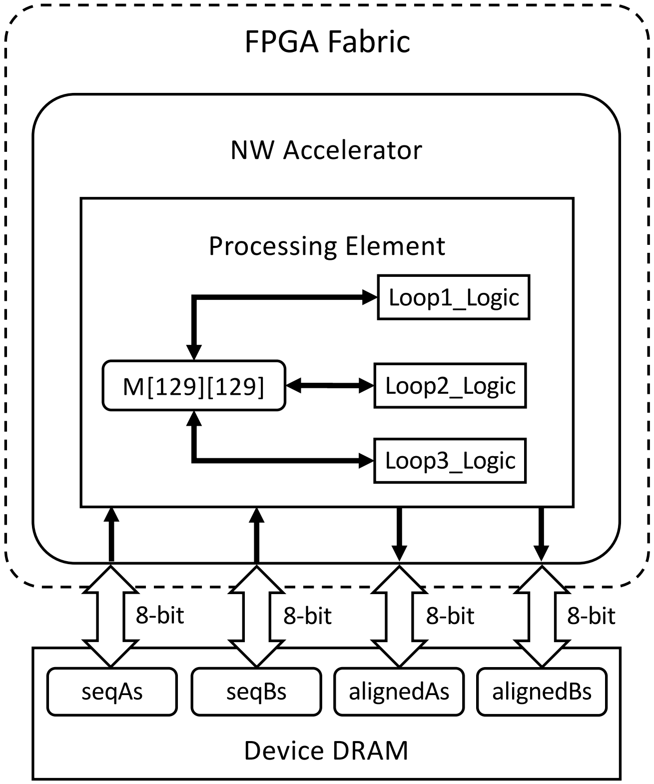
```
void engine(...) {
  int M[129][129];
  ...
loop1: for(i=0; i<129; i++) {M[0][i]=...}
loop2: for(j=0; j<129; j++) {M[j][0]=...}
loop3: for(i=1; i<129; i++) {
    for(j=1; j<129; j++) {...
      M[i][j]=...
  }}
  ...
}
void kernel(char seqAs[], char seqBs[],
       char alignedAs[], char alignedBs[]) {
  for (int i=0; i<NUM_PAIRS; i++) {
    engine(seqAs+i*128, seqBs+i*128,
       alignedAs+i*256, alignedBs+i*256);
}}
```



Limited Ports # of On-chip Buffers

How Can We Make it Work?

```
void engine(...) {
  int M[129][129];
  ...
loop1: for(i=0; i<129; i++) {M[0][i]=...}
loop2: for(j=0; j<129; j++) {M[j][0]=...}
loop3: for(i=1; i<129; i++) {
    for(j=1; j<129; j++) {...
      M[i][j]=...
  }}
  ...
}
void kernel(char seqAs[], char seqBs[],
       char alignedAs[], char alignedBs[]) {
  for (int i=0; i<NUM_PAIRS; i++) {
    engine(seqAs+i*128, seqBs+i*128,
       alignedAs+i*256, alignedBs+i*256);
}}
```



How Can We Make it Work?

Data transfer (DRAM & BRAM)

```
void buffer_load(
  int flag,
  double *global_in,
  double local_in[PE][JOBS_PER_PE * VECTOR_LENGTH],
  int *global_l,
  int local_l[PE][JOBS_PER_PE]) {
#pragma HLS INLINE off
  if (flag) {
    for (int i = 0; i < PE; i++) {
      memcpy(local_in[i], global_in + i * VECTOR_LENGTH * JOBS_PER_PE,
VECTOR_LENGTH * JOBS_PER_PE * 8);
      memcpy(local_l[i], global_l + i * JOBS_PER_PE, JOBS_PER_PE * 4);
    }
  }
  return ;
}
void buffer_store(int flag, int *global_out, int local_out[PE][JOBS_PER_PE]) {
#pragma HLS INLINE off
  if (flag) {
    for (int i = 0; i < PE; i++)
      memcpy(global_out + i * JOBS_PER_PE, local_out[i], JOBS_PER_PE * 4);
  }
```

Coarse-grained parallelism

```
void buffer_compute(
  int flag,
  double in[PE][JOBS_PER_PE * VECTOR_LENGTH],
  int len[PE][JOBS_PER_PE],
  int out[PE][JOBS_PER_PE]) {
#pragma HLS INLINE off
  if (flag) {
    for (int i = 0; i < PE; i++)
#pragma HLS UNROLL
      ProcessUnit(in[i], len[i], out[i]);
  }
  return ;
}
return ;
}
```

Computation

```
void ProcessUnit(double *in, int *lens, int *out) {
   // Original N-W function,
   // roughly 70 lines of code
}
```

```
void argmax(int N, double *in, int *lengths, int *output)
{
#pragma HLS INTERFACE m_axi port=in offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=lengths offset=slave bundle=gmem1
#pragma HLS INTERFACE m_axi port=output offset=slave bundle=gmem2
#pragma HLS INTERFACE s_axilite port=N bundle=control
#pragma HLS INTERFACE s_axilite port=in bundle=control
#pragma HLS INTERFACE s_axilite port=lengths bundle=control
#pragma HLS INTERFACE s_axilite port=output bundle=control
#pragma HLS INTERFACE s_axilite port=return bundle=control
```

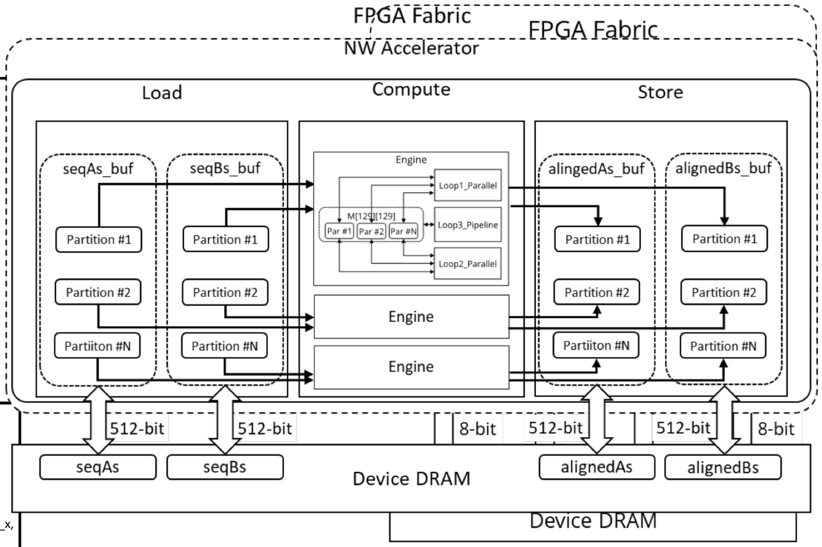
```
  double buf_in_x[PE][JOBS_PER_PE * VECTOR_LENGTH];
#pragma HLS ARRAY_PARTITION variable=buf_in_x complete dim=1
  double buf_in_y[PE][JOBS_PER_PE * VECTOR_LENGTH];
#pragma HLS ARRAY_PARTITION variable=buf_in_y complete dim=1
  double buf_in_z[PE][JOBS_PER_PE * VECTOR_LENGTH];
#pragma HLS ARRAY_PARTITION variable=buf_in_z complete dim=1

  int buf_l_x[PE][JOBS_PER_PE];
#pragma HLS ARRAY_PARTITION variable=buf_l_x complete dim=1
  int buf_l_y[PE][JOBS_PER_PE];
#pragma HLS ARRAY_PARTITION variable=buf_l_y complete dim=1
  int buf_l_z[PE][JOBS_PER_PE];
#pragma HLS ARRAY_PARTITION variable=buf_l_z complete dim=1

  int buf_out_x[PE][JOBS_PER_PE];
#pragma HLS ARRAY_PARTITION variable=buf_out_x complete dim=1
  int buf_out_y[PE][JOBS_PER_PE];
#pragma HLS ARRAY_PARTITION variable=buf_out_y complete dim=1
  int buf_out_z[PE][JOBS_PER_PE];
#pragma HLS ARRAY_PARTITION variable=buf_out_z complete dim=1
```

```
  int num_batches = N / JOBS_PER_BATCH;

  for (int i = 0; i < num_batches + 2; i++) {
   int load_flag = i >= 0 && i < num_batches;
   int compute_flag = i >= 1 && i < num_batches+1;
   int store_flag = i >= 2 && i < num_batches+2;
   if (i % 3 == 0) {
    buffer_load(load_flag, in + i * VECTOR_LENGTH * JOBS_PER_BATCH, buf_in_x,
lengths + i * JOBS_PER_BATCH, buf_l_x);
    buffer_compute(compute_flag, buf_in_z, buf_l_z, buf_out_z);
    buffer_store(store_flag, output + (i - 2) * JOBS_PER_BATCH, buf_out_y);
   }
   else if (i % 3 == 1) {
    buffer_load(load_flag, in + i * VECTOR_LENGTH * JOBS_PER_BATCH, buf_in_y,
lengths + i * JOBS_PER_BATCH, buf_l_y);
    buffer_compute(compute_flag, buf_in_x, buf_l_x, buf_out_x);
    buffer_store(store_flag, output + (i - 2) * JOBS_PER_BATCH, buf_out_z);
   }
   else if (i % 3 == 2) {
    buffer_load(load_flag, in + i * VECTOR_LENGTH * JOBS_PER_BATCH, buf_in_z,
lengths + i * JOBS_PER_BATCH, buf_l_z);
    buffer_compute(compute_flag, buf_in_y, buf_l_y, buf_out_y);
    buffer_store(store_flag, output + (i - 2) * JOBS_PER_BATCH, buf_out_x);
   }
  }
  return;
}
```



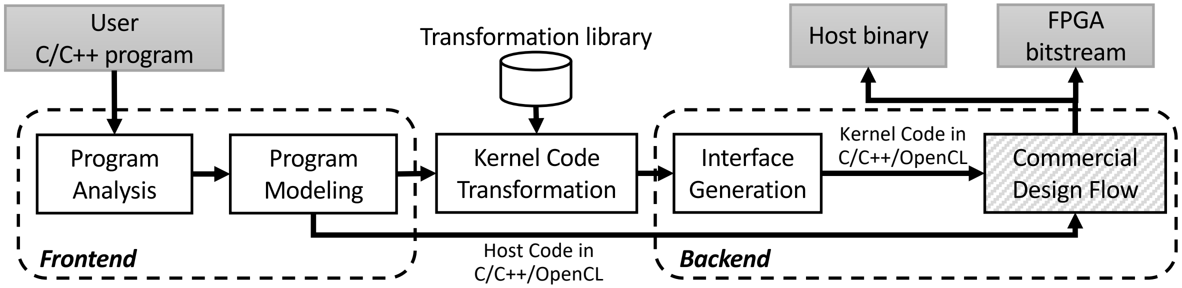
Coarse-grain pipeline

>1,000x speedup over single thread CPU!

…but also lots of efforts (~200 lines)!

Merlin Compiler: Simplify Code Reconstruction

- Overview



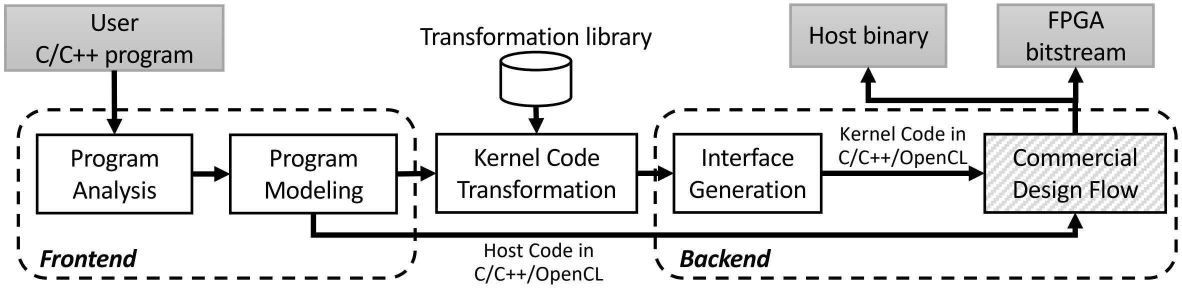
- Pragma-based transformations (similar to OpenMP)

Merlin Pragmas	Description	Vivado HLS
parallel	**Coarse-grained:** Wrap the computation to a function for HLS to generate PEs	Require code reconstruction
	Fine-grained: Partition array properly	Require manual memory partition
	Reduction: Construct a reduction tree	Require code reconstruction
pipeline	**Coarse-grained:** Create load-compute-store pipeline to overlap data transfer and compute	Require code reconstruction
	Fine-grained: Fully unroll all sub-loops if needed	Supported

More coarse-grained transformations compared to commercial HLS tools

Merlin Compiler: Simplify Code Reconstruction

- Overview


User C/C++ program

Transformation library

Host binary

FPGA bitstream

Program Analysis

Program Modeling

Kernel Code Transformation

Interface Generation

Kernel Code in C/C++/OpenCL

Commercial Design Flow

Host Code in C/C++/OpenCL

Frontend

Backend

- Example: simply add 3 pragmas to achieve the same performance

```
void kernel(int N, char seqA[], char seqB[],
            char outA[], char outB[]) {

#pragma ACCEL parallel=64
#pragma ACCEL pipeline
  for (int i=0; i<N; i++) {
    engine(seqA+i*128, seqB+i*128,
           outA+i*256, outB+i*256);
  }
}
```

```
void buffer_load(
  int flag,
  double *global_in,
  double local_in[PE][JOBS_PER_PE * VECTOR_LENGTH],
  int *global_l,
  int local_l[PE][JOBS_PER_PE]) {
#pragma HLS INLINE off
  if (flag) {
   for (int i = 0; i < PE; i++) {
     memcpy(local_in[i], global_in + i * VECTOR_LENGTH * JOBS_PER_PE,
VECTOR_LENGTH * JOBS_PER_PE * 8);
     memcpy(local_l[i], global_l + i * JOBS_PER_PE, JOBS_PER_PE * 4);
   }
  }
  return ;
}
void buffer_store(int flag, int *global_out, int local_out[PE][JOBS_PER_PE]) {
#pragma HLS INLINE off
  if (flag) {
   for (int i = 0; i < PE; i++)
     memcpy(global_out + i * JOBS_PER_PE, local_out[i], JOBS_PER_PE * 4);
  }
void buffer_compute(
  int flag,
  double in[PE][JOBS_PER_PE * VECTOR_LENGTH],
  int len[PE][JOBS_PER_PE],
  int out[PE][JOBS_PER_PE]) {
#pragma HLS INLINE off
  if (flag) {
   for (int i = 0; i < PE; i++)
#pragma HLS UNROLL
     ProcessUnit(in[i], len[i], out[i]);
  }
  return ;
}
return ;
}
void ProcessUnit(double *in, int *lens, int *out) {
   // Original N-W function,
   // roughly 70 lines of code
}
```

```
void argmax(int N, double *in, int *lengths, int *output)
{
#pragma HLS INTERFACE m_axi port=in offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=lengths offset=slave bundle=gmem1
#pragma HLS INTERFACE m_axi port=output offset=slave bundle=gmem2
#pragma HLS INTERFACE s_axilite port=N bundle=control
#pragma HLS INTERFACE s_axilite port=in bundle=control
#pragma HLS INTERFACE s_axilite port=lengths bundle=control
#pragma HLS INTERFACE s_axilite port=output bundle=control
#pragma HLS INTERFACE s_axilite port=return bundle=control

  double buf_in_x[PE][JOBS_PER_PE * VECTOR_LENGTH];
#pragma HLS ARRAY_PARTITION variable=buf_in_x complete dim=1
  double buf_in_y[PE][JOBS_PER_PE * VECTOR_LENGTH];
#pragma HLS ARRAY_PARTITION variable=buf_in_y complete dim=1
  double buf_in_z[PE][JOBS_PER_PE * VECTOR_LENGTH];
#pragma HLS ARRAY_PARTITION variable=buf_in_z complete dim=1

  int buf_l_x[PE][JOBS_PER_PE];
#pragma HLS ARRAY_PARTITION variable=buf_l_x complete dim=1
  int buf_l_y[PE][JOBS_PER_PE];
#pragma HLS ARRAY_PARTITION variable=buf_l_y complete dim=1
  int buf_l_z[PE][JOBS_PER_PE];
#pragma HLS ARRAY_PARTITION variable=buf_l_z complete dim=1

  int buf_out_x[PE][JOBS_PER_PE];
#pragma HLS ARRAY_PARTITION variable=buf_out_x complete dim=1
  int buf_out_y[PE][JOBS_PER_PE];
#pragma HLS ARRAY_PARTITION variable=buf_out_y complete dim=1
  int buf_out_z[PE][JOBS_PER_PE];
#pragma HLS ARRAY_PARTITION variable=buf_out_z complete dim=1

  int num_batches = N / JOBS_PER_BATCH;

  for (int i = 0; i < num_batches + 2; i++) {
   int load_flag = i >= 0 && i < num_batches;
   int compute_flag = i >= 1 && i < num_batches+1;
   int store_flag = i >= 2 && i < num_batches+2;
   if (i % 3 == 0) {
     buffer_load(load_flag, in + i * VECTOR_LENGTH * JOBS_PER_BATCH, buf_in_x,
lengths + i * JOBS_PER_BATCH, buf_l_x);
     buffer_compute(compute_flag, buf_in_z, buf_l_z, buf_out_z);
     buffer_store(store_flag, output + (i - 2) * JOBS_PER_BATCH, buf_out_y);
   }
   else if (i % 3 == 1) {
     buffer_load(load_flag, in + i * VECTOR_LENGTH * JOBS_PER_BATCH, buf_in_y,
lengths + i * JOBS_PER_BATCH, buf_l_y);
     buffer_compute(compute_flag, buf_in_x, buf_l_x, buf_out_x);
     buffer_store(store_flag, output + (i - 2) * JOBS_PER_BATCH, buf_out_z);
   }
   else if (i % 3 == 2) {
     buffer_load(load_flag, in + i * VECTOR_LENGTH * JOBS_PER_BATCH, buf_in_z,
lengths + i * JOBS_PER_BATCH, buf_l_z);
     buffer_compute(compute_flag, buf_in_y, buf_l_y, buf_out_y);
     buffer_store(store_flag, output + (i - 2) * JOBS_PER_BATCH, buf_out_x);
   }
  }
  return;
}
```

Available from Falcon Computing: https://www.falconcomputing.com

HLS Challenges and Solutions

- Challenge 1: Heavy code reconstruction
  - Modern HLS tools require particular coding style for performance
  - ***Solution: The Merlin compiler***
- Challenge 2: Large design space
  - Should we use coarse-grained pipeline?
  - What parallel factor should we use for each loop?
  - How to determine on-chip buffer sizes?

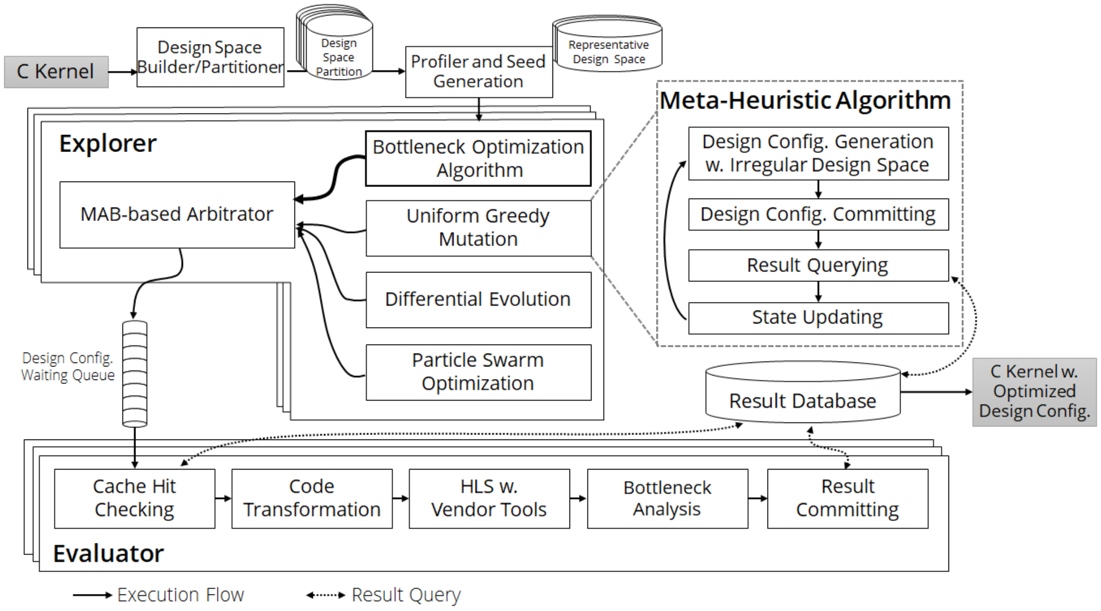
Automated Design Space Exploration Framework

Design Space

- A general design space representation

Search Approach

- Multi-armed bandit approach with meta-heuristics
- Gradient-based approach with design bottleneck analysis



C Kernel

Design Space Builder/Partitioner

Design Space Partition

Profiler and Seed Generation

Representative Design Space

Explorer

MAB-based Arbitrator

Bottleneck Optimization Algorithm

Uniform Greedy Mutation

Differential Evolution

Particle Swarm Optimization

Meta-Heuristic Algorithm

Design Config. Generation w. Irregular Design Space

Design Config. Committing

Result Querying

State Updating

Design Config. Waiting Queue

Result Database

C Kernel w. Optimized Design Config.

Cache Hit Checking

Code Transformation

HLS w. Vendor Tools

Bottleneck Analysis

Result Committing

Evaluator

Execution Flow

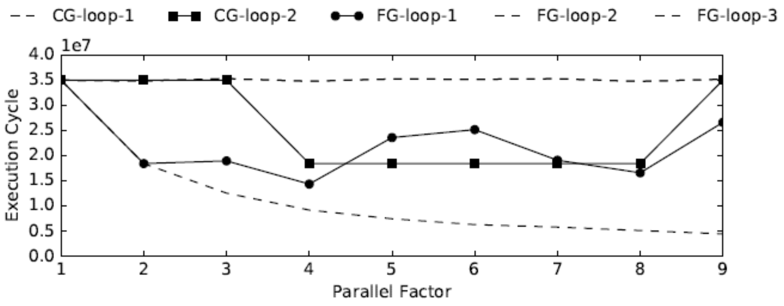
Result Query

Evaluation Methodology

Evaluate the design quality using commercial HLS tools

Gradient Search Approach

- Toward to the single-move design point according to gradient
    - $Gradient \sim FiniteDifference = \frac{\Delta Latency}{\Delta Resource\ Util.}$
- Guarantee to improve QoR every iteration
- Challenges
    - Unpredictable HLS tool behavior
    - Serious local optimal problem
    - Long evaluation time (30 mins – 1 hr)

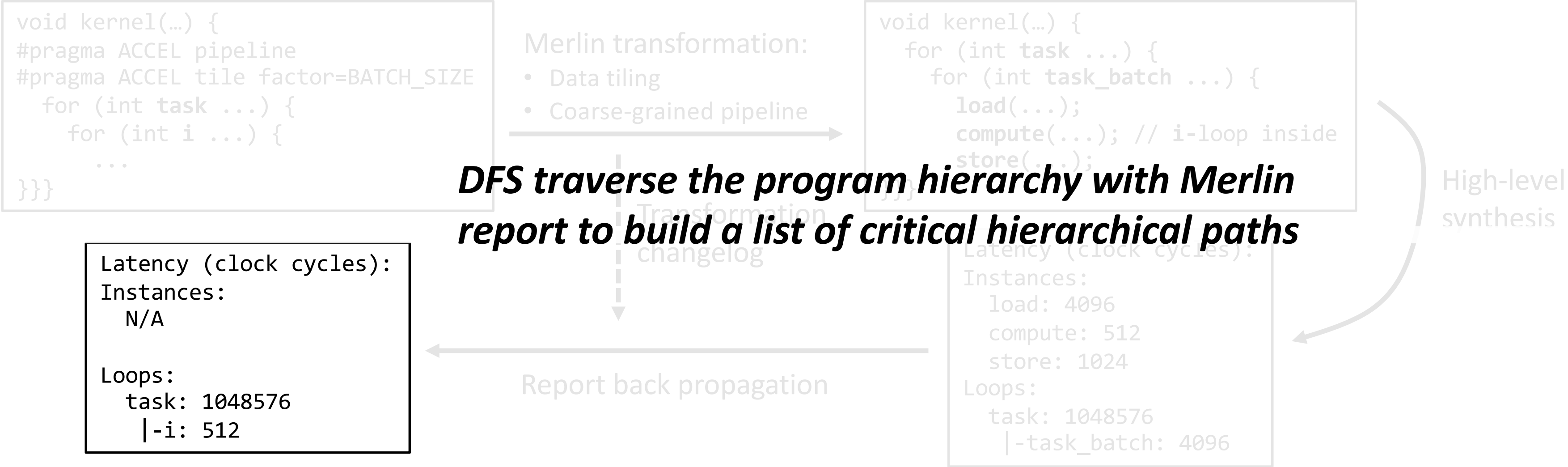


Strategies to Avoid Local Optimal inspired by VLSI Physical Design

- Design space partition
  - Separate design points with huge QoR different
- Adaptive line search
  - Try the option that may not result in weird QoR (e.g., power of two factors)
- Multi-scale V-cycle
  - Group the parameters that should be explored together and release them later

Design Bottleneck Analysis

- Performance bottleneck analysis with Merlin performance report



```
void kernel(…) {
#pragma ACCEL pipeline
#pragma ACCEL tile factor=BATCH_SIZE
  for (int task ...) {
    for (int i ...) {
      ...
}}}
```

```
void kernel(…) {
  for (int task ...) {
    for (int task_batch ...) {
      load(...);
      compute(...); // i-loop inside
      store(...);
}}}
```

```
Latency (clock cycles):
Instances:
  load: 4096
  compute: 512
  store: 1024
Loops:
  task: 1048576
   |-task_batch: 4096
```

```
Latency (clock cycles):
Instances:
  N/A

Loops:
  task: 1048576
   |-i: 512
```

DFS traverse the program hierarchy with Merlin report to build a list of critical hierarchical paths

- Gradient-based search approach improvement
  - Identify a small set of critical parameters by bottleneck analysis
  - Parallel explore the factors of the critical parameter to avoid local optimal

HLS Challenges and Solutions

- Challenge 1: Heavy code reconstruction
  - Modern HLS tools require particular coding style for performance
  - ***Solution: The Merlin compiler***
- Challenge 2: Large design space
  - Should we use coarse-grained pipeline?
  - What parallel factor should we use for each loop?
  - How to determine on-chip buffer sizes?
  - ***Solution: Automated design space exploration***

Experimental Results

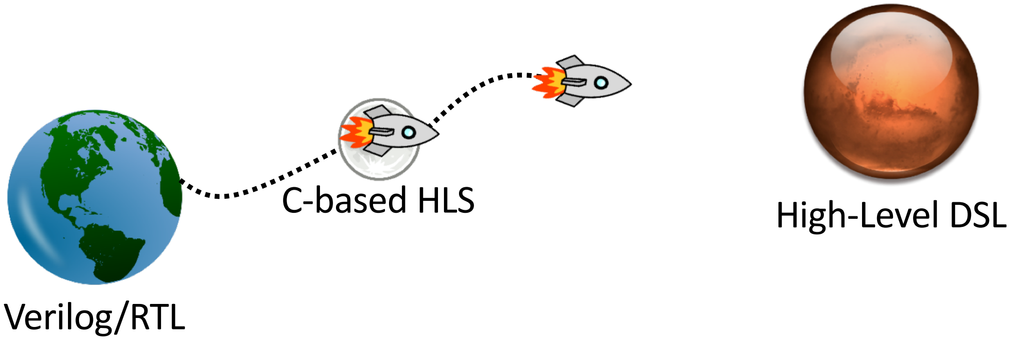
- Configuration
  - Amazon EC2 F1 instance (`f1.2xlarge`) with 8-core CPU and 122 GB memory
  - Xilinx Vertex UltraScale+TM VU9P FPGA
  - 4 hour DSE with 8 threads
- Benchmark: Machsuite, RodiniaUCLA, AlexNet
  - Baseline: Single-thread CPU
  - Reference: Manual optimization with Merlin pragmas
- Results
  - 11/12 cases achieve >80% manual performance

Benchmark	Design Space	Ratio to Manual (%)	Speedup over CPU
AES	3.11E+09	**100%**	3774.69
NW	1.51E+09	**97.67%**	3387.46
KMP	5.76E+03	52.24%	5.04
GEMM	1.26E+09	**100%**	16.25
SPMV	5.76E+03	**100%**	1.73
STENCIL-2D	9.70E+09	**94.00%**	0.39
STENCIL-3D	1.94E+06	**100%**	2.65
BACKPROP	1.15E+04	**100%**	7.71
KMEANS	2.49E+05	**99.18%**	34.82
KNN	1.90E+04	**99.84%**	9.48
PATHFINDER	5.18E+03	**88.62%**	0.16
CONV	1.50E+28	**93.96%**	55.06
Geometric Mean	**1.26E+08**	**93.78%**	**13.69**

2nd place (w. necessary code change) in 51 submissions of UCLA CS133

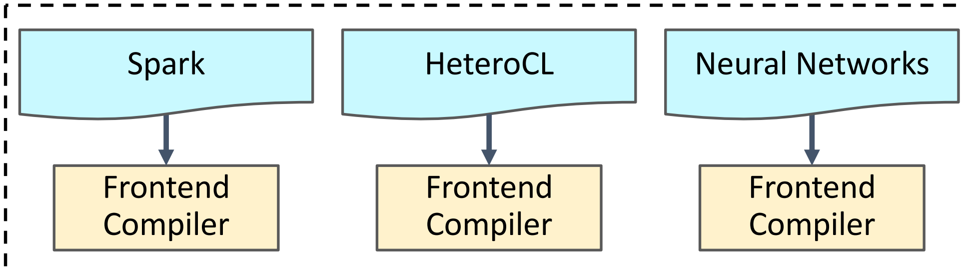
Higher Level Abstraction -- Domain-Specific Languages (DSL) Support?

- We are now traveling to the Mars!

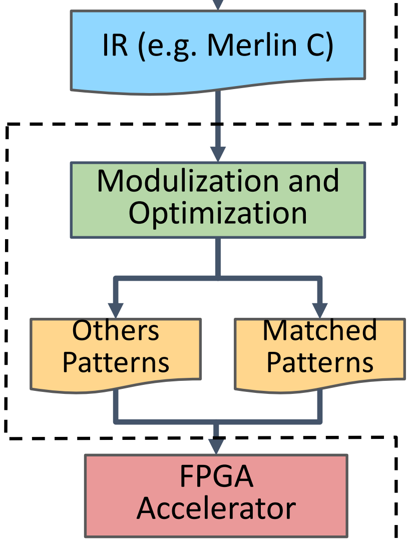


- Advantages of raising the abstraction level to DSLs
  - Expend the usability and accessibility of FPGAs
  - Further improve the programmability
  - Clearer scheduling information to achieve better performance

From Domain-Specific Languages (DSLs) to FPGAs

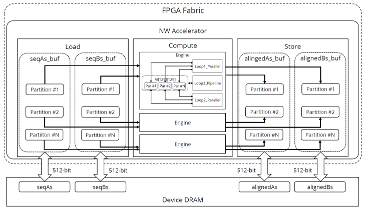


Frontend: DSLs to Merlin C

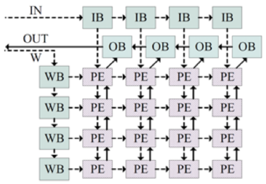


Matched patterns

- Model-based DSE
- Pre-defined architectures



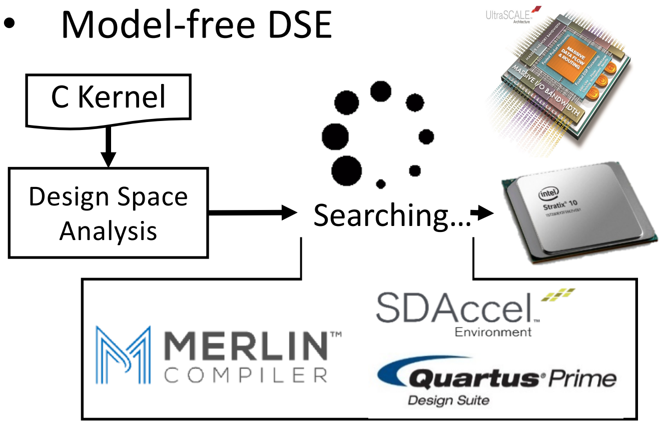
Stencil
[ICCAD '18]



Systolic Array
[DAC '17, ICCAD '18]

Unmatched patterns

- Arbitrary architecture
- Model-free DSE



Backend: Optimization

DSL Synthesis Challenges

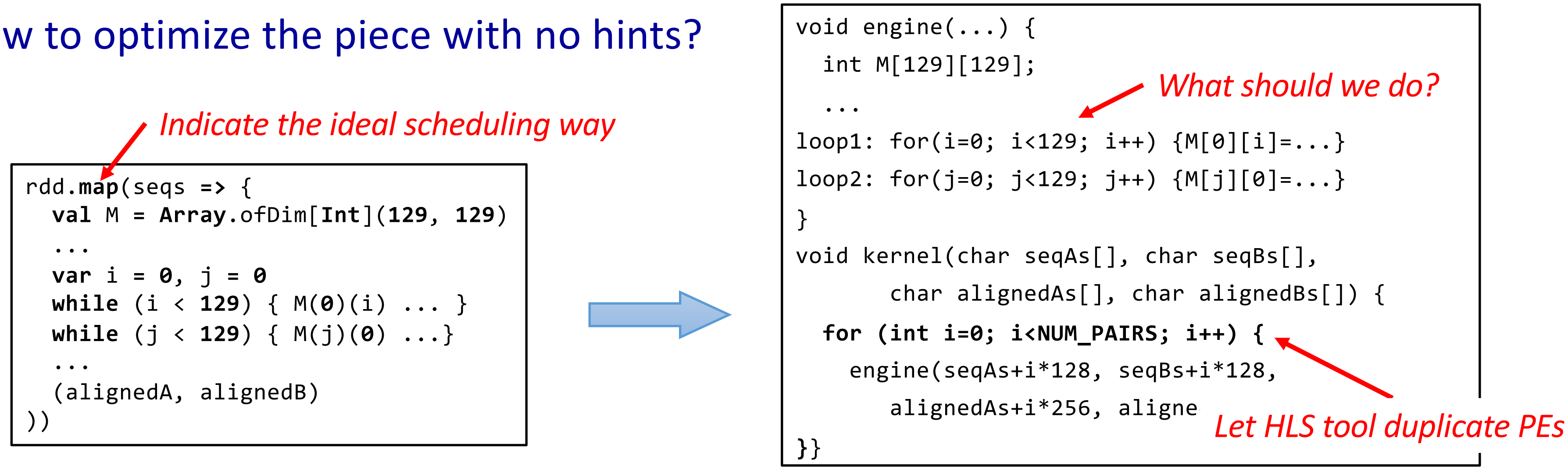
- Challenge 1: Semantic transferring (functionality)
  - A DSL-to-C compiler that translates syntax while preserving the semantics

```
rdd.map(seqs => {
  val M = Array.ofDim[Int](129, 129)
  ...
  var i = 0, j = 0
  while (i < 129) { M(0)(i) ... }
  while (j < 129) { M(j)(0) ...}
  ...
  (alignedA, alignedB)
))
```

```
void engine(...) {
  int M[129][129];
  ...
loop1: for(i=0; i<129; i++) {M[0][i]=...}
loop2: for(j=0; j<129; j++) {M[j][0]=...}
}
void kernel(char seqAs[], char seqBs[],
       char alignedAs[], char alignedBs[]) {
  for (int i=0; i<NUM_PAIRS; i++) {
    engine(seqAs+i*128, seqBs+i*128,
       alignedAs+i*256, alignedBs+i*256);
}}
```

More DSL Synthesis Challenges

- Challenge 1: Semantic equivalence
- Challenge 2: Design pattern preservation (opportunity)
  - Perverse as many DSL information as possible to help tuning performance
  - How to reflect all scheduling “hints” to the generated HLS code?
  - How to optimize the piece with no hints?



Indicate the ideal scheduling way

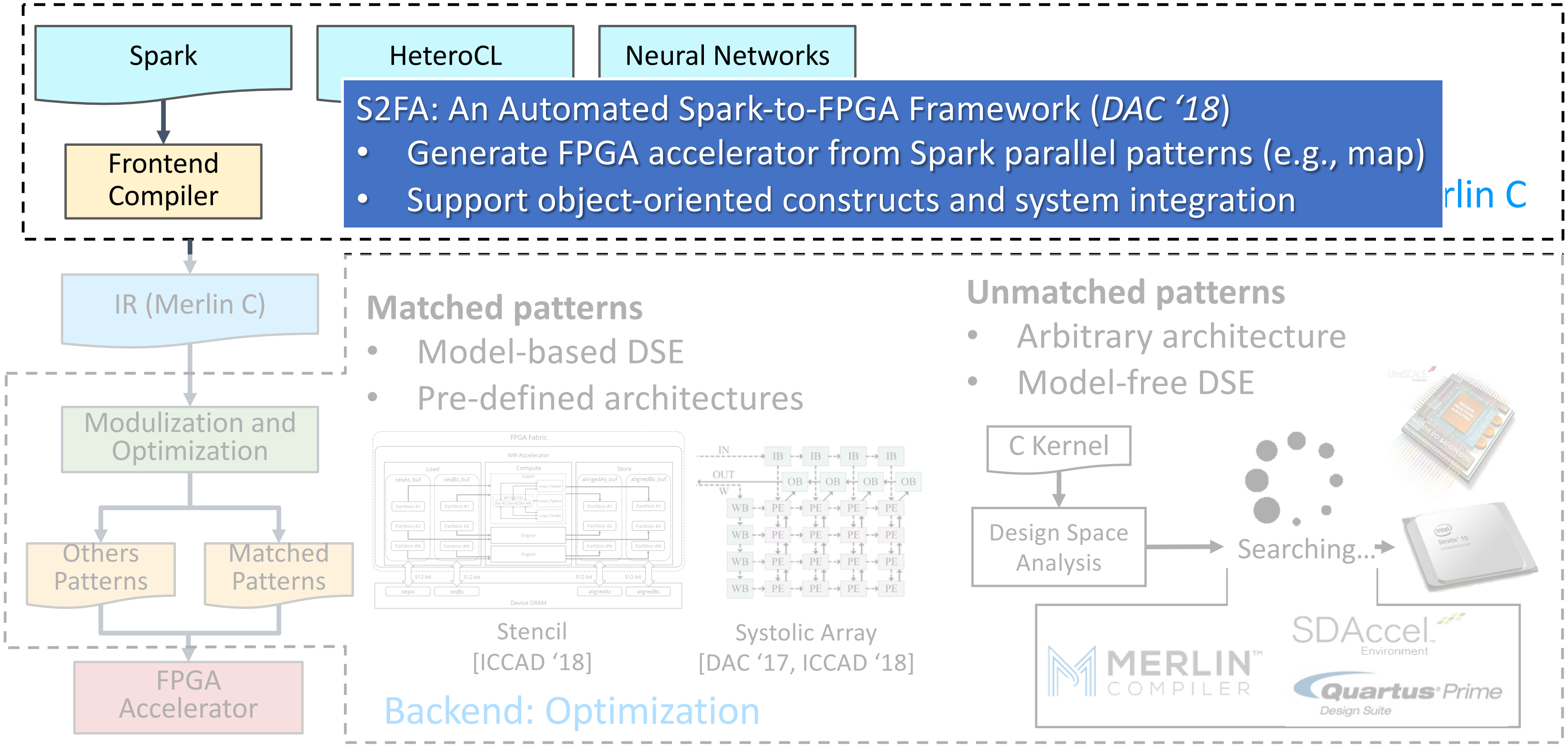
```
rdd.map(seqs => {
  val M = Array.ofDim[Int](129, 129)
  ...
  var i = 0, j = 0
  while (i < 129) { M(0)(i) ... }
  while (j < 129) { M(j)(0) ...}
  ...
  (alignedA, alignedB)
))
```

What should we do?

```
void engine(...) {
  int M[129][129];
  ...
loop1: for(i=0; i<129; i++) {M[0][i]=...}
loop2: for(j=0; j<129; j++) {M[j][0]=...}
}
void kernel(char seqAs[], char seqBs[],
       char alignedAs[], char alignedBs[]) {
  for (int i=0; i<NUM_PAIRS; i++) {
    engine(seqAs+i*128, seqBs+i*128,
       alignedAs+i*256, aligne
}}
```

Let HLS tool duplicate PEs

Example 1: From DSL to FPGAs



S2FA: An Automated Spark-to-FPGA Framework (*DAC ‘18*)
- Generate FPGA accelerator from Spark parallel patterns (e.g., map)
- Support object-oriented constructs and system integration

Spark

HeteroCL

Neural Networks

Frontend Compiler

rlin C

IR (Merlin C)

Modulization and Optimization

Others Patterns

Matched Patterns

FPGA Accelerator

Matched patterns
- Model-based DSE
- Pre-defined architectures

Stencil [ICCAD ‘18]

Systolic Array [DAC ‘17, ICCAD ‘18]

Unmatched patterns
- Arbitrary architecture
- Model-free DSE

C Kernel

Design Space Analysis

Searching…

Backend: Optimization

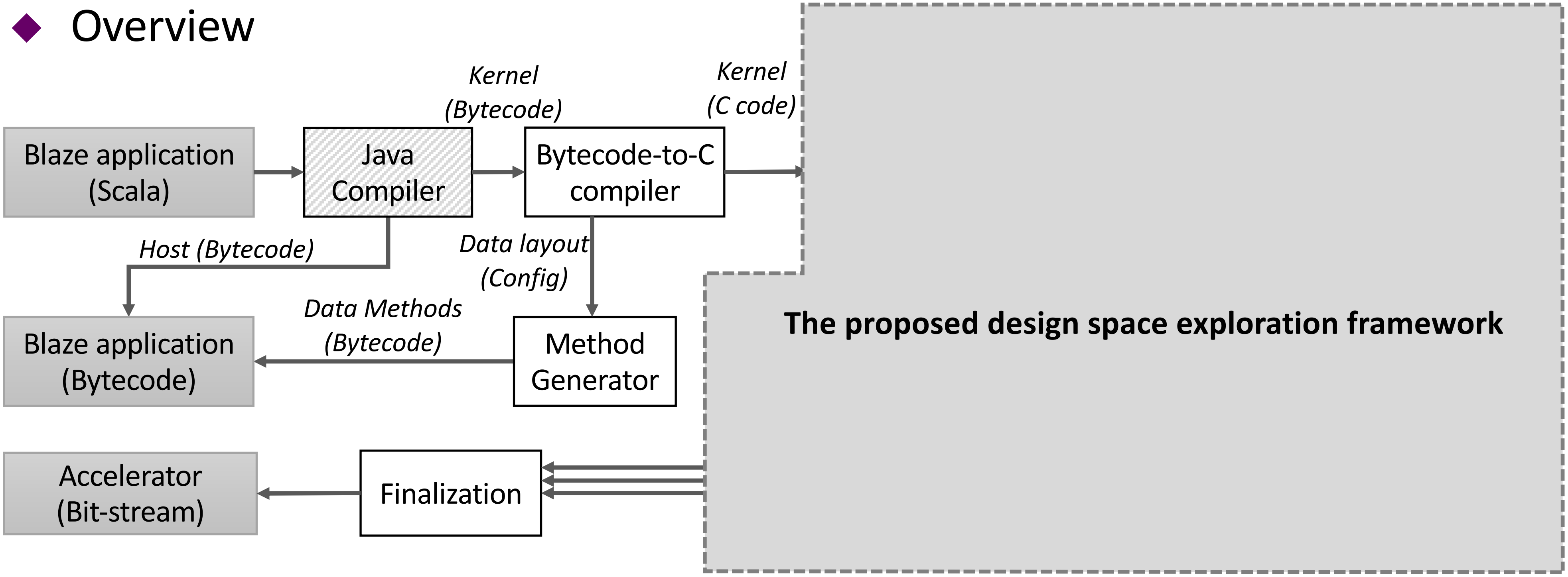
S2FA Framework Overview

- Programming model

Java annotation to provide necessary information

```
@S2FA_Kernel(Vector.values:128)
def call(seqA: Vector, seqB: Vector) = { ... }
```

- Overview



Blaze application (Scala)

Java Compiler

Kernel (Bytecode)

Bytecode-to-C compiler

Kernel (C code)

Host (Bytecode)

Data layout (Config)

Blaze application (Bytecode)

Data Methods (Bytecode)

Method Generator

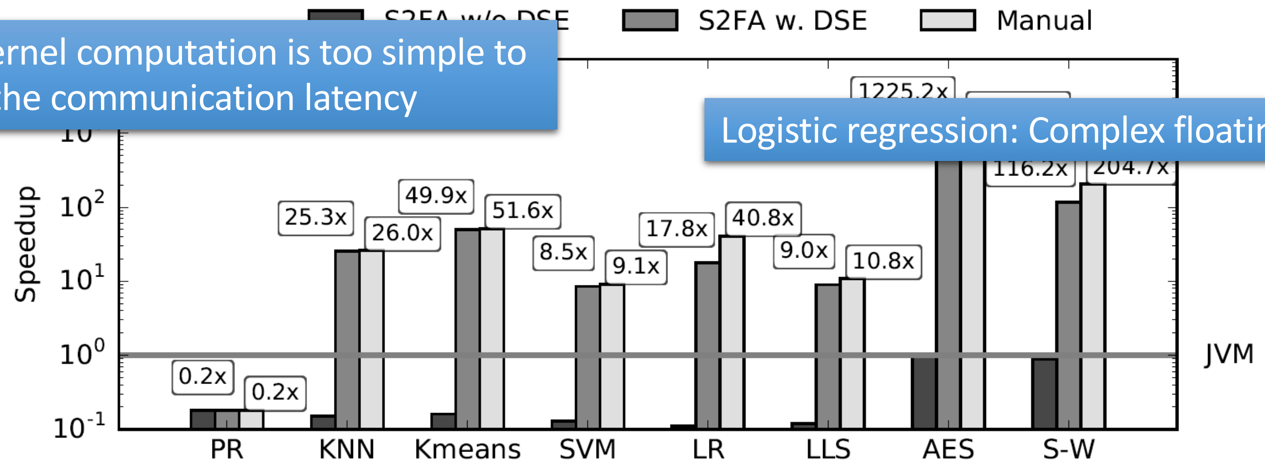
Accelerator (Bit-stream)

Finalization

The proposed design space exploration framework

S2FA Evaluation Results

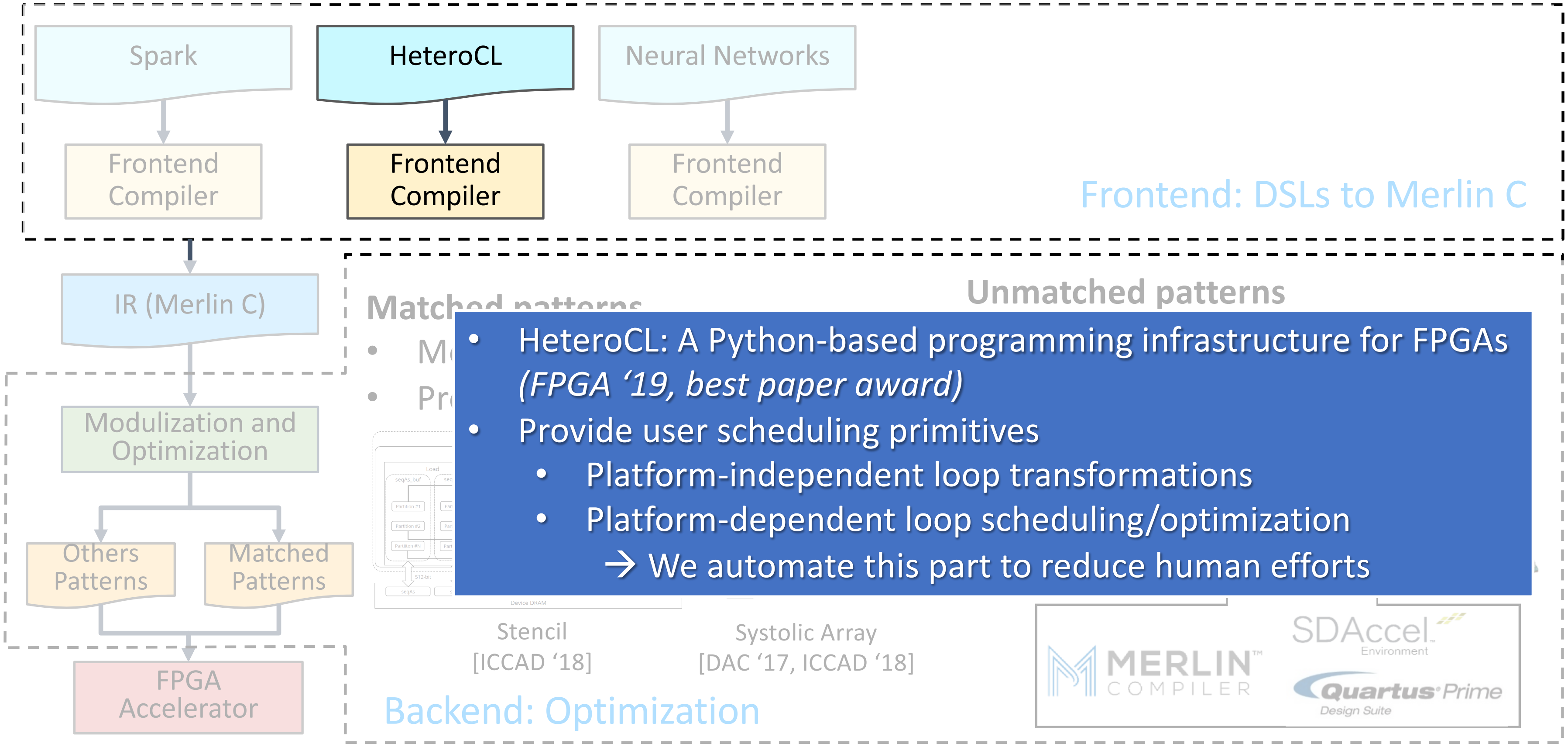
- Platform
  - Amazon EC2 F1 instance (`f1.2xlarge`) with Xilinx Vertex UltraScale+^{TM} VU9P FPGA
- Results
  - Achieve 181.5x performance over the baseline (single-thread JVM)
  - Achieve 85% performance on average of manual designs



PageRank: the kernel computation is too simple to hide the communication latency

Logistic regression: Complex floating-point computation

Example 2: From DSL to FPGAs

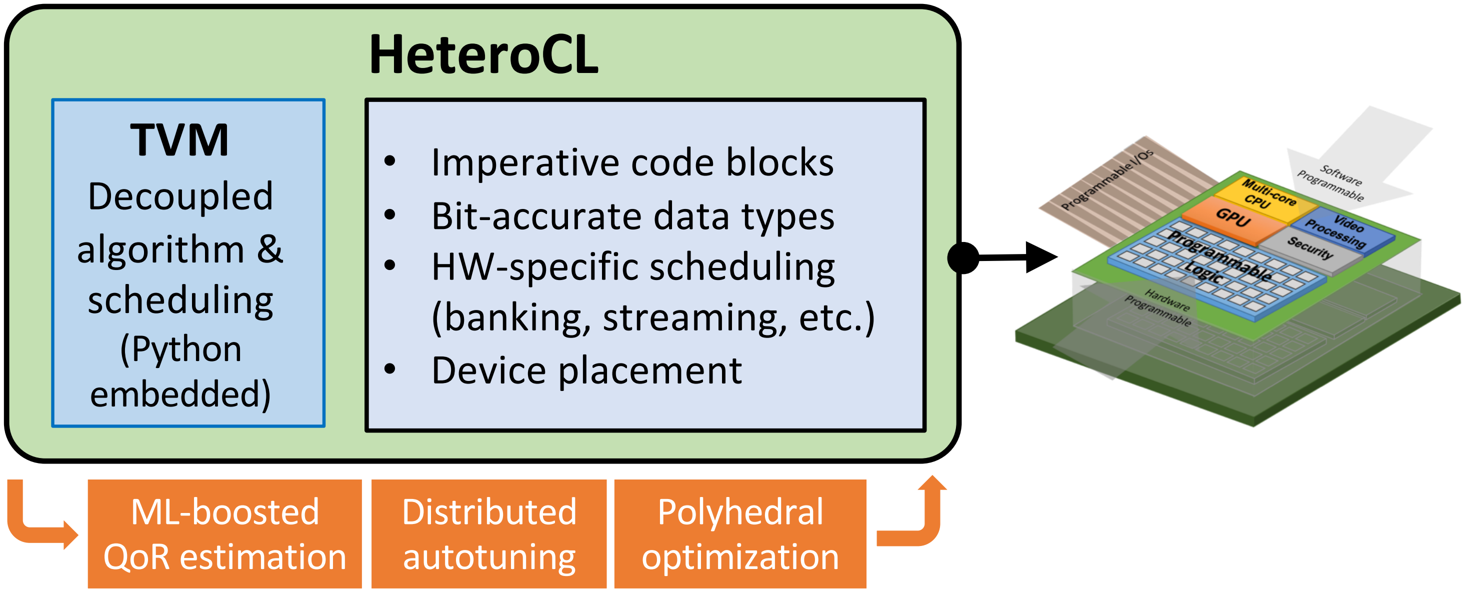


- HeteroCL: A Python-based programming infrastructure for FPGAs *(FPGA '19, best paper award)*
- Provide user scheduling primitives
  - Platform-independent loop transformations
  - Platform-dependent loop scheduling/optimization
    → We automate this part to reduce human efforts

HeteroCL Programming Model (Joint Work between Cornell & UCLA)

- A novel intermediate language that explicitly exposes heterogeneity in three dimensions
  - in programing model with mixed declarative and imperative code
  - in optimization with decoupled algorithm and compute/data customization
  - in hardware targets with flexible code and data placement

Open source: https://vast.cs.ucla.edu/software/heterocl
https://github.com/cornell-zhang/heterocl



HeteroCL

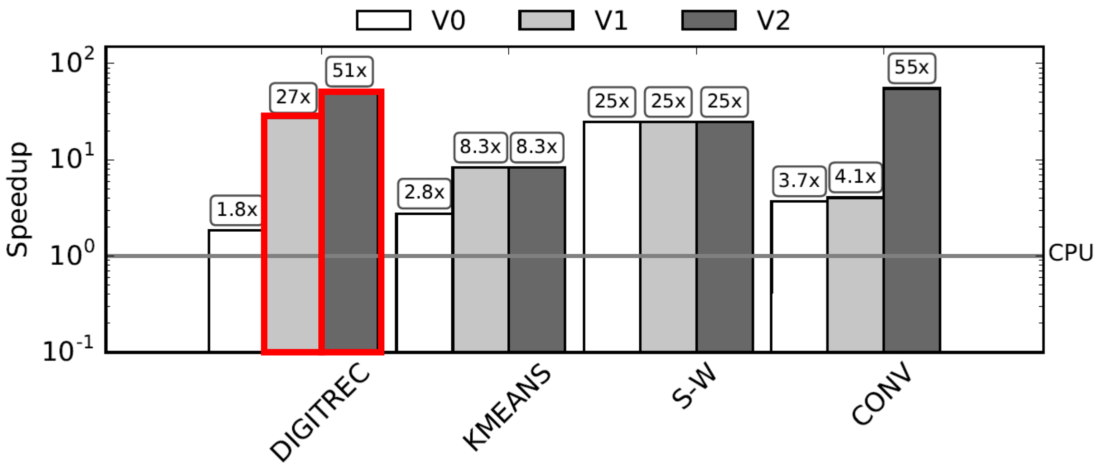
TVM
Decoupled algorithm & scheduling (Python embedded)

- Imperative code blocks
- Bit-accurate data types
- HW-specific scheduling (banking, streaming, etc.)
- Device placement

ML-boosted QoR estimation | Distributed autotuning | Polyhedral optimization

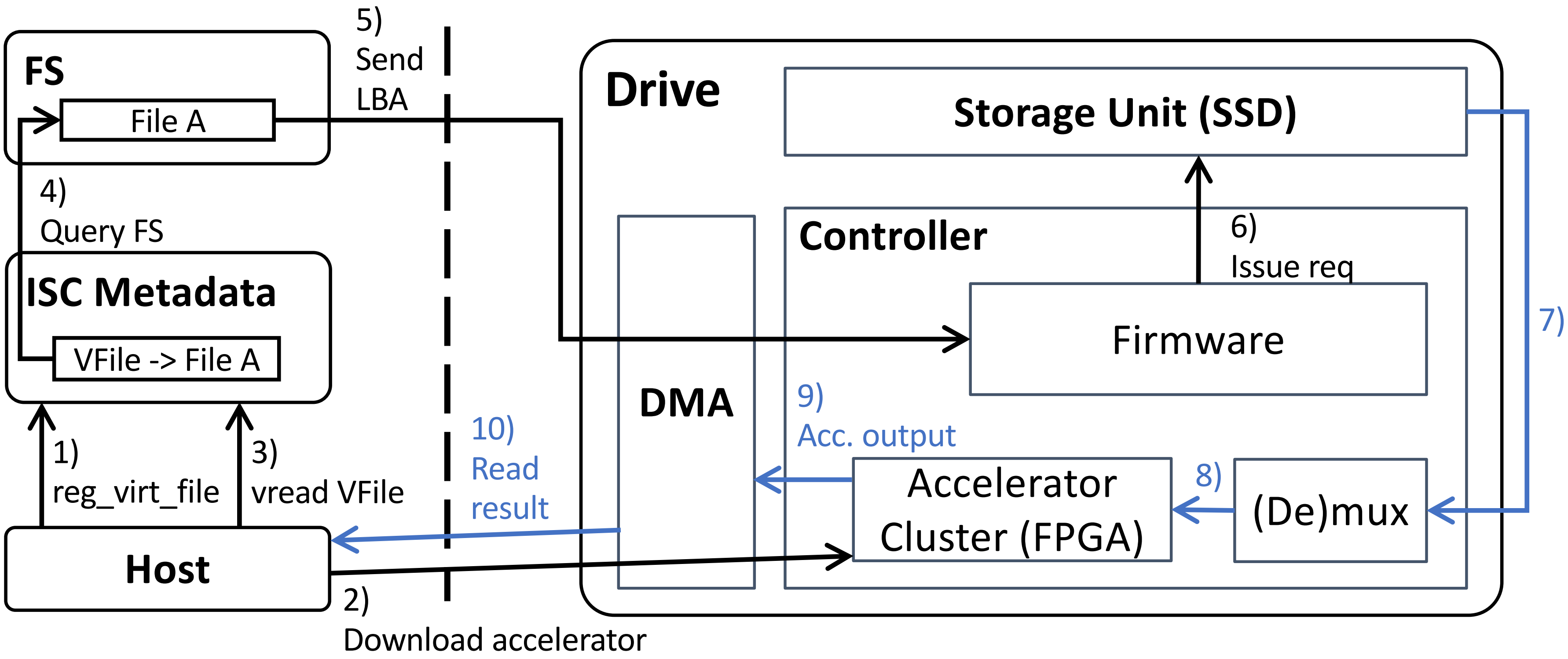
Initial Auto-HeteroCL Results

- Platform
  - Amazon EC2 F1 instance (`f1.2xlarge`) with 8-core CPU and 122 GB memory
  - Xilinx Vertex UltraScale+^{TM} VU9P FPGA
- Gradually apply loop transformation scheduling primitives with DSE



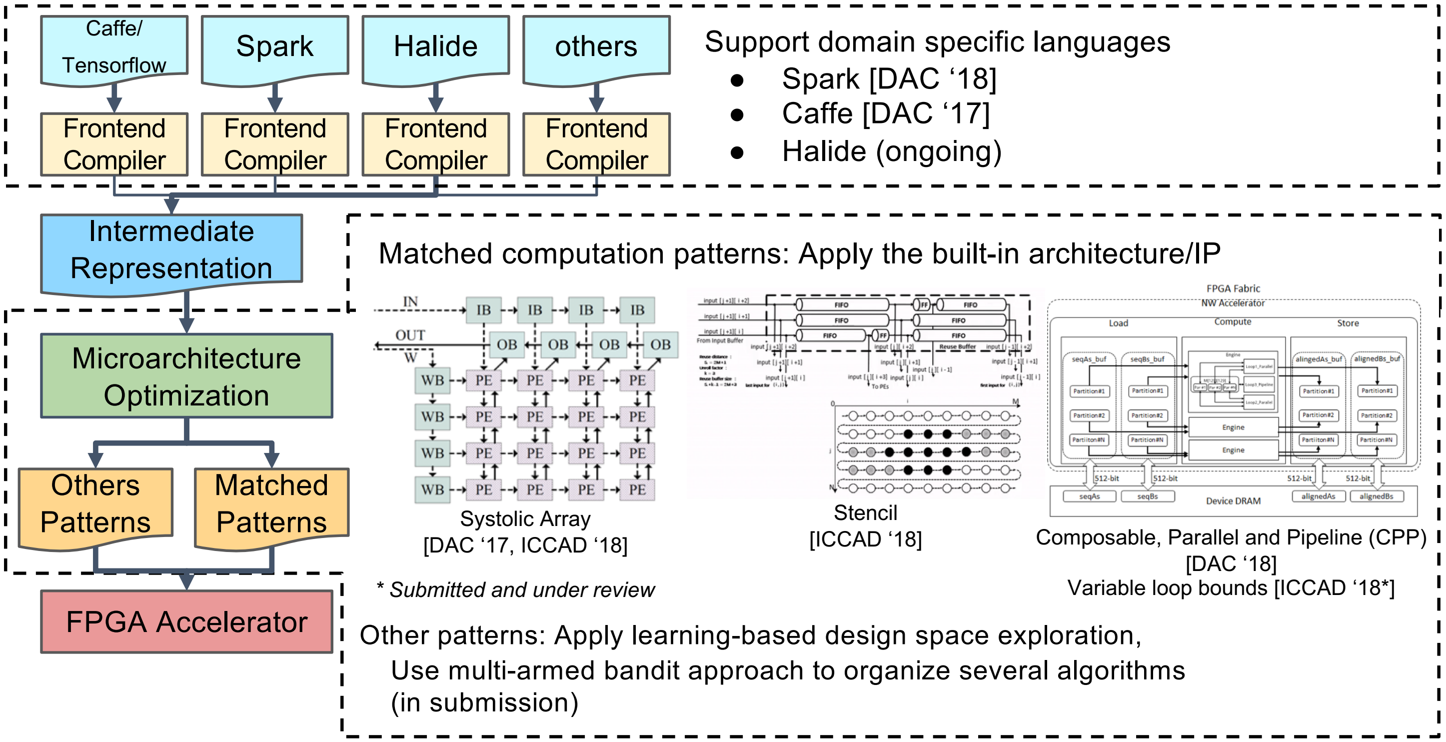
Design	V1	V2
DIGITREC	+Loop Merging	+Loop reorder
KMEANS	+Loop reorder	N/A
S-W	N/A	N/A
CONV	+Loop Splitting	+Loop reorder

Another Example: Support of In-Storage Acceleration [ATC'2019]



Summary –
Democratization of Customization by Better Automations & Higher Level of Abstraction

Good progress, a lot more to be done!



Support domain specific languages

- Spark [DAC '18]
- Caffe [DAC '17]
- Halide (ongoing)

Matched computation patterns: Apply the built-in architecture/IP

Systolic Array
[DAC '17, ICCAD '18]

Stencil
[ICCAD '18]

Composable, Parallel and Pipeline (CPP)
[DAC '18]
Variable loop bounds [ICCAD '18*]

** Submitted and under review*

Other patterns: Apply learning-based design space exploration,
Use multi-armed bandit approach to organize several algorithms
(in submission)

Goal: You innovate (in algorithm, application …),
we automate (compiling to customized hardware)

Acknowledgements: NSF, CRISP, and CDSC Industrial Partners

Multi-year Efforts by Students, Postdocs, and Collaborators


Yuze Chi
(UCLA)

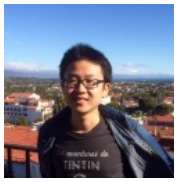

Young-kyu Choi
(UCLA)


Prof. Miryung Kim
(UCLA)


Prof. Louis-Noël
Pouchet
(UCLA/colostate)


Prof. Adrian Sampson
(Cornell Univ.)


Prof. Vivek Sarkar
(Georgia Tech)


Jie Wang
(UCLA)


Yi-Hsiang Lai
(Cornell)


Yuxin Wang
(PKU/Falcon)


Peng Wei
(UCLA)


Di Wu
(UCLA/Falcon)


Hao Yu
(UCLA/Falcon)


Dr. Peng Zhang
(UCLA/Falcon)


Prof. Zhiru Zhang
(Cornell Univ.)


Peipei Zhou
(UCLA)


Yuan Zhou (Cornell)