Democratize Customizable Computing

Jason Cong
Computer Science Department, UCLA
June 2019
High-level synthesis for FPGA Programming is Real

- Here is the moon!

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

However, it’s not an easy journey
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

```c
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

```c
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);
  }
}
Not All C Programs Lead to Good Performance

Example: The Needleman-Wunsch algorithm for sequence alignment

```c
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);
    }
}
Not All C Programs Lead to Good Performance

Example: The Needleman-Wunsch algorithm for sequence alignment

```c
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<NUMPAIRS; i++) {
        engine(seqAs+i*128, seqBs+i*128,
            alignedAs+i*256, alignedBs+i*256);
    }
```
How Can We Make it Work?

```c
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);
    }
}
```
Coarse-grained parallelism

Data transfer
(DRAM & BRAM)

Computation

How Can We Make it Work?

void buffer_load() {
    for (int i = 0; i < num_batches + 2; i++) {
        buffer_load(i, in, out); // Original N-W function,
        // roughly 70 lines of code
    }
}

int num_batches = N / JOBS_PER_BATCH;
for (int i = 0; i < num_batches + 2; i++) {
    int load_flag = 0;
    int compute_flag = 0;
    int store_flag = 0;
    if (i == 0) {
        load_flag = 1;
        compute_flag = 1;
        store_flag = 0;
    } else if (i == num_batches + 1) {
        load_flag = 0;
        compute_flag = 0;
        store_flag = 1;
    } else {
        load_flag = 0;
        compute_flag = 1;
        store_flag = 0;
    }
    buffer_load(i, in, out, load_flag, compute_flag, store_flag);
}

>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)

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

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

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

```c
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);
    }
}
```

Available from Falcon Computing: [https://www.falconcomputing.com](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

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 $= \frac{\Delta\text{Latency}}{\Delta\text{Resource Util.}}$
- 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**

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

  **Merlin transformation:**
  - Data tiling
  - Coarse-grained pipeline

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

  **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+™ 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

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

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

**Unmatched patterns**
- Arbitrary architecture
- Model-free DSE

**Modulization and Optimization**

**Others Patterns**
- Matched Patterns

**FPGA Accelerator**

**Backend: Optimization**

- **Stercil** [ICCAD ‘18]
- **Systolic Array** [DAC ‘17, ICCAD ‘18]
**DSL Synthesis Challenges**

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

```
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?

```java
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, alignedB);
    }
}
```
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

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

Unmatched patterns
- Arbitrary architecture
- Model-free DSE

Frontend: DSLs to Merlin C

Backend: Optimization

Merlin C
S2FA Framework Overview

- Programming model

  ```
  @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)

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+™ 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

Frontend: DSLs to Merlin C

- Spark
- HeteroCL
- Neural Networks
- Frontend Compiler

IR (Merlin C)

Modulization and Optimization

Others Patterns

Matched Patterns

- 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

Unmatched patterns

- Arbitrary architecture
- Model-free DSE

C Kernel

Design Space Analysis

Searching…

Backend: Optimization

- Stencil [ICCAD ‘18]
- Systolic Array [DAC ‘17, ICCAD ‘18]

FPGA Accelerator

HeteroCL

Frontend Compiler

Frontend Compiler

Frontend Compiler

Frontend Compiler
HeteroCL Programming Model (Joint Work between Cornell & UCLA)

- A novel intermediate language that explicitly exposes heterogeneity in three dimensions
  - in programming 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

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

TVM
Decoupled algorithm & scheduling (Python embedded)

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+™ VU9P FPGA

Gradually apply loop transformation scheduling primitives with DSE

![Speedup Graph]

<table>
<thead>
<tr>
<th>Design</th>
<th>V1</th>
<th>V2</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIGITREC</td>
<td>+Loop Merging</td>
<td>+Loop reorder</td>
</tr>
<tr>
<td>KMEANS</td>
<td>+Loop reorder</td>
<td>N/A</td>
</tr>
<tr>
<td>S-W</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>CONV</td>
<td>+Loop Splitting</td>
<td>+Loop reorder</td>
</tr>
</tbody>
</table>
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*]

- **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)

Yuexin 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)