High-Level Synthesis and Beyond
-- from Datacenters to IoTs

Jason Cong
Chancellor’s Professor, UCLA
Director, Center for Domain-Specific Computing
Co-Director, PKU/UCLA Joint Research Institute in Science and Engineering
cong@cs.ucla.edu
http://vast.cs.ucla.edu/people/faculty/jason-cong
PLATFORM-BASED BEHAVIOR-LEVEL AND
SYSTEM-LEVEL SYNTHESIS

Jason Cong, Yiping Fan, Guoling Han, Wei Jiang, Zhiru Zhang
University of California, Los Angeles
Email: cong, fanyp, leohgl, wijiang, zhiruzc@cs.ucla.edu

Abstract—With the rapid increase of complexity in System-on-a-Chip (SoC) design, the electronic design automation (EDA) community
is moving from RTL (Register Transfer Level) synthesis to behavioral-level and system-level synthesis. The needs of system-level verification and software/hardware co-design also
prefer behavior-level executable specifications, such as C or SystemC. In this paper we present the platform-based synthesis system, named XPlot, being developed at UCLA. The
first objective of XPlot is to provide novel behavioral synthesis capability for automatically generating efficient RTL code from a C or SystemC description
for a given system platform and optimizing the logic, interconnects, performance, and power simultaneously. The second objective of XPlot is to provide
a platform-based system-level synthesis capability, including both synthesis for application-specific configurable processors and heterogeneous
multi-core systems. Preliminary experiments on FPGAs demonstrate the eficacy of our approach on a wide range of applications and its value in exploring various design tradeoffs.

I. MOTIVATION

The relentless tracking of Moore’s curve by the entire semiconductor industry has led to the exponential scaling of the transistor feature size by a factor of 0.7 reduction every three years. This leads to exponentially increasing transistor counts and results in an exponential growth in functionality and the amount of computing power available on a single chip. Today it is perfectly feasible to design a System-on-a-Chip (SoC) with one billion transistors [7], and it is generally believed that industry will continue to overcome technical hurdles to sustain this trend for another decade. However, the cost of developing these chips and providing production facilities is also growing at a very fast pace. For instance, the total development cost of a single complex, high-density SoC at today’s 90-nm technology can easily be in the $20 to $30 million range. The ITRS 2005 edition [7] has also emphasized that the cost of design remains the greatest threat to continuation of the semiconductor roadmap.

Unfortunately, the progress of design technologies lags behind that of process manufacturing technologies. The constantly improving CAD tools can help to mitigate the problem by delivering faster simulation, higher capacity formal verification, and better logic synthesis coupled with place-and-route. However, these improvements fail to close the design productivity gap, i.e., the number of available transistors grows faster than the ability to meaningfully design them.

It is commonly acknowledged that the ultimate solution is to move to the next level of abstraction beyond RTL and Electronic system-level (ESL) design automation has been widely identified as the next productivity boost for the semiconductor industry. However, despite some recent success in ESL simulation, the transition to ESL design will not be as well accepted as the transition to RTL without robust and efficient behavior-level and system-level synthesis technologies that automatically synthesize high-level functional descriptions into optimized software/hardware implementations. We believe that behavior-level and system-level synthesis and optimizations are becoming important steps in EDA design flows. They provide the following combined advantages:

Better complexity management: Design abstraction is one of the most effective methods for controlling rising complexity and improving design productivity. For example, a recent study from NEC [8] shows that the code density (in terms of line counts) can be improved by nearly 10X when moved to the behavior level. In addition, behavior-level and system-level synthesis have the added value of allowing efficient reuse of soft functional/behavioral IPs, which are technology-independent and can be synthesized for different requirements.

Shorter verification/simulation cycle: System-level synthesis and optimizations allow the designers to start with a specification in a high-level programming language (HLP) such as C or SystemC that is directly executable and simulatable with high speed (up to 1000X faster than RTL-level simulation according to [8]). More importantly, behavioral synthesis automatically compiles the input descriptions into RTL code through a series of formal constructive transformations. This avoids the slow and error-prone manual process and simplifies the design verification and debugging effort.

Rapid system exploration: With the coexistence of microprocessors, DSPs, memories and custom logic on a single chip, more software elements are involved in the process of designing a modern embedded system. One of the fundamental challenges of system-level design is the hardware/software partitioning, a task that is too complex to be feasible at the RTL level. HLP-based design methodologies (especially C-based designs) offer a promising solution to this problem. With the aid of behavior-level synthesis, the software programming languages can also be used to specify functionality in hardware. In this flow, designers can quickly experiment with different hardware/software boundaries by co-simulating the HLP descriptions and the automatically synthesized HDLs.

Higher quality of results: VLSI designers in current semi-conductor technologies are limited by interconnect in both delay and power. However, since the interconnects are deter-
C/C++ to FPGA Synthesis

xPilot (UCLA) -> AutoPilot (AutoESL) -> Vivado HLS (Xilinx)

- Platform-based C to RTL synthesis
- Synthesize pure ANSI-C and C++, GCC-compatible compilation flow
- 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

Developed by AutoESL, acquired by Xilinx in Jan. 2011
• Wireless MIMO Sphere Decoder
  – ~4000 lines of C code
  – Xilinx Virtex-5 at 225MHz
• Compared to optimized IP
  – 11-31% better resource usage

TCAD April 2011 (keynote paper)
“High-Level Synthesis for FPGAs: From Prototyping to Deployment”
AutoPilot Results: Optical Flow (from BDTI)

- **Application**
  - Optical flow, 1280x720 progress scan
  - Design too complex for an RTL team

- **Compared to high-end DSP:**
  - 30X higher throughput, 40X better cost/fps

<table>
<thead>
<tr>
<th>Chip Unit Cost</th>
<th>Highest Frame Rate @ 720p (fps)</th>
<th>Cost/Performance ($/frame/second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xilinx Spartan3ADSP XC3SD3400A chip</td>
<td>$27</td>
<td>183</td>
</tr>
<tr>
<td>Texas Instruments TMS320DM6437 DSP processor</td>
<td>$21</td>
<td>5.1</td>
</tr>
</tbody>
</table>

BDTi evaluation of AutoPilot
http://www.bdti.com/articles/AutoPilot.pdf
Learn More about AutoESL/AutoPilot

www.autoe.nl

High-Level Synthesis for FPGAs: From Prototyping to Deployment

Jason Cong, Fellow, IEEE, Bin Liu, Stephen Neuendorffer, Member, IEEE, Juanjo Nogueira, Kees Vissers, Members, IEEE, and Zhiyu Zhang, Member, IEEE

Abstract—Escalating system-on-chip design complexity is pushing the design community to raise the level of abstraction beyond register transfer level. Despite the success of early generations of commercial high-level synthesis (HLS) systems, we believe that the tipping point for transitioning to HLS methodology is happening now, especially for field-programmable gate array (FPGA) designs. The latest generation of HLS tools has made significant progress in providing wide language coverage and robust compilation technology, platform-based modeling, advancement in core HLS algorithms, and a domain-specific approach. In this paper, we use AutoESL’s AutoPilot HLS tool coupled with domain-specific system-level implementation platform developed by Xilinx as an example to demonstrate the effectiveness of state-of-the-art C-to-FPGA synthesis solutions targeting multiple application domains. Complex industrial designs targeting Xilinx FPGAs are also presented as case studies, including comparison of HLS solutions versus optimized manual designs. In particular, the experiment on a sphere decoder shows that the HLS solution can achieve an 11–31% reduction in FPGA resource usage with improved design productivity compared to hand-coded design.

Index Terms—Domain-specific design, field-programmable gate array (FPGA), high-level synthesis (HLS), quality of results (QoR).

1. INTRODUCTION

The rapid increase of complexity in system-on-chip (SoC) design has encouraged the design community to seek design abstractions with better productivity than register transfer level (RTL). Electronic system-level (ESL) design automation has been widely identified as the main productivity boost for the semiconductor industry, where high-level synthesis (HLS) plays a central role, enabling the automatic synthesis of high-level, untimed or partially timed specifications (such as in C or SystemC) to low-level cycle-accurate RTL specifications for efficient implementation in application-specific integrated circuits (ASICs) or field-programmable gate arrays (FPGAs). This synthesis can be optimized taking into account the performance, power, and cost requirements of a particular system.

Despite the past failure of the early generations of commercial HLS systems (started in the 1990s), we see a rapidly growing demand for innovative, high-quality HLS solutions for the following reasons.

1) Embedded processors are in almost every SoC: With the coexistence of micro-processors, digital signal processors (DSPs), memories and custom logic on a single chip, more software elements are involved in the process of designing a modern embedded system. An automated HLS flow allows designers to specify design functionality in high-level programming languages such as C/C++, for both embedded software and customized hardware logic on the SoC. This way, they can quickly experiment with different hardware/software boundaries and explore various area/power/performance tradeoffs from a single common functional specification.

2) Huge Silicon capacity requires a higher level of abstraction: Design abstraction is one of the most effective methods for controlling complexity and improving design productivity. For example, the study from NEC [91] shows that a 1M-gate design typically requires about 300K lines of RTL code, which cannot be easily handled by a human designer. However, the code density can be easily reduced by 7X–10X when moved to high-level specification in C, C++, or SystemC. In this case, the same 1M-gate design can be described in 30K–60K lines of behavioral description, resulting in a much reduced design complexity.

3) Behavioral IP reuse improves design productivity: In addition to the line-count reduction in design specifications, behavioral synthesis has the added value of allowing efficient reuse of behavioral intellectual properties (IPs). As opposed to RTL IP which has fixed microarchitecture and interface protocols, behavioral IP can be retargeted to different implementation technologies or system requirements.

4) Verification drives the acceptance of high-level specification: Transaction-level modeling (TLM) with SystemC [109] or similar C/C++ based extensions has become a
Continued Success at Xilinx (after 2011 acquisition): Vivado High-Level Synthesis (HLS) for Hardware IP Creation

Accelerate Algorithmic C to IP Integration
- Available in production today for C, C++, SystemC
  - Adopted across broad base of applications and markets
  - Proven on real customer designs
  - Clear differentiator for accelerating design productivity

Accelerate Algorithmic C to Co-Processing Accelerator Integration
“Vivado® HLS enabled easy and fast implementation of 768x768 QRD single precision floating-point design. We like this tool QoR, productivity and flexibility and will deploy in to more production designs.”

“Vivado HLS made it possible for DSP software engineers to implement LTE layer 1 switch on Zynq® SoCs by enabling us to target more than 500K lines of C code.”

“We developed C++ DSP functions using Vivado HLS and the results met size/speed goal for commercial platform deployment on Virtex®-7.”
C to Verified RTL from Months to Weeks

"In an **HDL** design, each scenario would likely cost an **additional day of writing code** …
With Vivado **HLS** these changes **took minutes**"

*Nathan Jachimiec, R&D Engineer, Agilent Technologies*

"I was able to design complex linear algebra algorithms **10x faster** than before with VHDL, and yet achieved **better QoR** with Vivado HLS."

*Design Engineer, Major A&D contractor*

"..we always use **C** to quickly build a **system-level model** for validation of **key algorithms**.. problem.. quickly and efficiently convert **C** into a **HDL**."

*Hengqi Liu, Central R&D Data Center CTO, ZTE Inc.*

<table>
<thead>
<tr>
<th>Radar Design</th>
<th>Conventional Hand-coded HDL Approach</th>
<th>Using Vivado High Level Synthesis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design Language</td>
<td>VHDL (RTL)</td>
<td>C</td>
</tr>
<tr>
<td>Design Time (weeks)</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>Latency (ms)</td>
<td>37</td>
<td>21</td>
</tr>
<tr>
<td>Memory (RAMB18E1)</td>
<td>134 (16%)</td>
<td>10 (1%)</td>
</tr>
<tr>
<td>Memory (RAMB36E1)</td>
<td>273 (65%)</td>
<td>138 (33%)</td>
</tr>
<tr>
<td>Registers</td>
<td>29686 (9%)</td>
<td>14263 (4%)</td>
</tr>
<tr>
<td>LUTs</td>
<td>28152 (18%)</td>
<td>24257 (16%)</td>
</tr>
</tbody>
</table>

*Source: Design Engineer at Major A&D contractor*

"For each project where we used Vivado HLS, we **saved 2-3 weeks** of engineering **time**."

*CTO, Major broadcast equipment company*
High-Level Synthesis Came a Long Way …

- **Early attempts**
  - Research projects
  - 1980s ~ early 1990s

- **Rise and fall of early commercialization**
  - Tools from major EDA vendors
  - 1990s~early 2000s

- **Renewed interests**
  - Start-ups, followed by major EDA vendors
  - Mid 2000 ~ present

- **Wide adoption by the FPGA design community**
  - Led by Xilinx Vivado HLS (based on AutoESL acquisition in 2011)
  - 2012 ~ present
What Made xPilot/AutoPilot/Vivado HLS Successful

♦ Use of LLVM compilation infrastructure
  ▪ A good decision made in 2004

♦ Platform-based synthesis
  ▪ RTLs are optimized for different implementation platforms
    ● Cell and interconnect delays, memory configurations, I/O ports/types …

♦ Most importantly, algorithmic innovations
  ▪ Global optimization under multiple constraints, objectives, e.g.
    ● SDC based scheduling [DAC’06, ICCAD’13]
    ● Use of soft constraints and behavior-level don’t-cares [TODAES’10, BPA]
    ● Simultaneous register and FU binding [DATE’10]
    ● Automatic memory partitioning [TODAES’11, BPA] …
  ▪ Result: competitive to manual RTL designs
Now HLS has finally succeeded, what else?
Challenges and Opportunities Above and Beyond HLS

- Enable customized computing at datacenter scale
  - Better programming environment and compilation
  - Runtime management

- Enable rapid design and deployment of IoTs
### What’s Wrong with Processors?

<table>
<thead>
<tr>
<th></th>
<th>Throughput</th>
<th>Power</th>
<th>Figure of Merit (Gb/s/W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES 128bit key</td>
<td>3.84 Gbits/sec</td>
<td>350 mW</td>
<td>11 (1/1)</td>
</tr>
<tr>
<td>128bit data</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.18mm CMOS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPGA [1]</td>
<td>1.32 Gbit/sec</td>
<td>490 mW</td>
<td>2.7 (1/4)</td>
</tr>
<tr>
<td>ASM StrongARM [2]</td>
<td>31 Mbit/sec</td>
<td>240 mW</td>
<td>0.13 (1/85)</td>
</tr>
<tr>
<td>ASM Pentium III [3]</td>
<td>648 Mbits/sec</td>
<td>41.4 W</td>
<td>0.015 (1/800)</td>
</tr>
<tr>
<td>C Emb. Sparc [4]</td>
<td>133 Kbits/sec</td>
<td>120 mW</td>
<td>0.0011 (1/10,000)</td>
</tr>
<tr>
<td>Java [5] Emb. Sparc</td>
<td>450 bits/sec</td>
<td>120 mW</td>
<td>0.000037 (1/3,000,000)</td>
</tr>
</tbody>
</table>

[1] Amphion CS5230 on Virtex2 + Xilinx Virtex2 Power Estimator
[4] gcc, 1 mW/MHz @ 120 Mhz Sparc – assumes 0.25 u CMOS
[5] Java on KVM (Sun J2ME, non-JIT) on 1 mW/MHz @ 120 MHz Sparc – assumes 0.25 u CMOS

Various Improvements Are Not Enough

- Case study of H.264 [Hameed et al., ISCA’2010]
  - Optimization using SIMD + VLIW $\rightarrow$ 10x energy efficiency
  - Customized instruction fusion $\rightarrow$ 1.6x energy efficiency

- Still 50x away from ASICs – Not enough!
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.
How to Improve the Efficiency? Our Proposal – Customized Computing with Accelerator-Rich Architectures

- A customizable heterogeneous platform (CHP)
  - With a sea of dedicated and composable accelerators
  - Most computations are carried on accelerators – not on processors!

- A fundamental departure from von Neumann architecture

- Why now?
  - Previous architectures are device/transistor limited
  - Von Neumann architecture allows maximum device reuse
    - One pipeline serves all functions, fully utilized

- Future architectures
  - Plenty of transistors, but power/energy limited (dark silicon)
  - Customization and specialization for maximum energy efficiency

- A story of specialization
Lessons from Nature: Human Brain and Advance of Civilization

- High power efficiency (20W) of human brain comes from specialization
  - Different region responsible for different functions
- Remarkable advancement of civilization also from specialization
  - More advanced societies have higher degree of specialization
Levels of Customization

- **Single-chip level**
  - Require new processor designs, e.g. using composable accelerators [ISLPED’12, DAC’14]

- **Server node level**
  - Host CPU + FPGA via PCI-e or QPI connections

- **Data center level**
  - Clusters of heterogeneous computing nodes
Levels of Customization

◆ Single-chip level
  ▪ Require new processor designs, e.g. using composable accelerators [ISLPED’12, DAC’14]

◆ Sever node level
  ▪ Host CPU + FPGA via PCI-e or QPI connections

◆ Data center level
  ▪ Clusters of heterogeneous computing nodes
1. Initial Scan
An ultra low dose scan of entire chest region is obtained for lung cancer screening (e.g., using 25% of normal dosage).

2. Image Reconstruction
The imaging study is reconstructed using compressed sensing methods, improving initial quality of the images relative to current methods. Accelerated platforms and techniques are used.

3. Automated Detection
Images are processed to detect suspicious nodules on-the-fly, assessing whether a focused scan is needed. Advanced methods for image segmentation, feature extraction, and classification are employed.

4. Adaptive Diagnostic Scan
If nodules are detected, a diagnostic follow-up scan is performed during the same visit, limited to the regions containing suspicious regions of interest. Again, iterative compressive sensing methods can be used, reducing dosage by up to 50% of current dosage.

5. Clinical Interpretation
Based on the focused diagnostic scan, a radiologist can provide a clinical interpretation for the study. Detected nodules are overlaid on the diagnostic imaging study.

If no nodules are seen on the low-dose CT study, the patient can go home without any further imaging required.

Success Example: Medical Imaging Processing Pipeline
Example of CDSC Heterogeneous Computing Server

“Commodity” Intel Server

Intel® Xeon® Processor

Intel® Memory Controller Hub (MCH)

Intel® I/O Subsystem

Standard Intel® x86-64 Server

x86-64 Linux

Convey FPGA-based coprocessor

Application Engine Hub (AEH)

Application Engines (AEs)

XC6vlx760 FPGAs
80GB/s off-chip bandwidth
94W Design Power

Convey coprocessor
FPGA-based
Shared cache-coherent memory

Direct Data Port

## 5 Years of Accelerating Medical Image Processing

<table>
<thead>
<tr>
<th></th>
<th>2010</th>
<th>2013</th>
<th>2015 (Today)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CT image reconstruction</strong></td>
<td>18 hours Single thread CPU</td>
<td>20 minutes FPGA acceleration on Convey</td>
<td>6 minutes 4 Virtex-6 FPGAs on Convey w/data reuse</td>
</tr>
<tr>
<td><strong>Denoising</strong></td>
<td>5 minutes Single thread CPU</td>
<td>15 seconds NVidia GPU</td>
<td>3 seconds Core i7 Haswell, OpenMP, stencils</td>
</tr>
<tr>
<td><strong>Registration</strong></td>
<td>10 minutes Single thread CPU</td>
<td>2 minutes NVidia GPU</td>
<td>30 seconds Core i7 Haswell, OpenMP, stencils</td>
</tr>
<tr>
<td><strong>Segmentation</strong></td>
<td>20 minutes Single thread CPU</td>
<td>4 minutes Multithread CPU</td>
<td>1 minute Core i7 Haswell, OpenMP, stencils</td>
</tr>
<tr>
<td><strong>Analysis</strong></td>
<td>45 minutes Single thread CPU</td>
<td>18 minutes Multithread CPU</td>
<td>5 minutes* Core i7 Haswell, OpenMP</td>
</tr>
</tbody>
</table>

* New detection method w/improved accuracy

---

![Workstation](image.png)  ![CPU, GPU, FPGA](image.png)  ![FPGA, CPU](image.png)
HLS Alone Is Not Enough

- A large design space for software and hardware co-design
- Also need automated source code transformation for HLS friendly C/C++ code to enable
  - Concurrent memory access
  - Data reuse and on-chip buffer generation
  - Data prefetching
  - ...

Design Complexity and Optimization Opportunities

dequant -> idct_row -> idct_col

transform coefficient
run/level[][]

decoder

motion vector[][]
decoded images[][]

texture
denoise

error[][]
prediction[][]
decoded images[][]
enhanced images[][]
Design Complexity and Optimization Opportunities

- Streaming vs. shared?
- FIFO vs. switching buffer?
- Sync. granularity?
- Address mapping?

- HW or SW?
- Implementation options?
- Duplication?
- System performance?

- Data prefetching?
- Data reuse?
- Buffer size vs. bandwidth?
- On-chip memory throughput?

- run/level[][]
- motion vector[][]
- decoded images[][]
- enhanced images[][]

parser

c

texture

denoise
CMOST: Fully Automated Compilation and Mapping Flow

Input: C/C++ w/ kernel/task pragmas  User Directives  Platform Spec.

Program Analysis
- Task graph extraction
- HW/SW partitioning
- Driver generation
- OpenCL generation
- Test generation

Design analysis/impl. report

System Optimization
- Module evaluation
- Data reuse
- Block streaming
- Prefetching
- Module selection & duplication

System Generation
- Module templates, System IP templates (C/RTL)
- Configure C/RTL/scripts
- Xilinx Vivado HLS
- Xilinx Vivado

On-board executable HW/SW

Retargetable and optimized OpenCL source code
**Optimization Beyond HLS**

**Input Code (C/C++)**
- **Program Analysis**
- **Loop Restructuring**
- **Code Generation**

**Loop Structure Optimization**
- Array Partitioning
- Data Reuse

**Data Layout Optimization**
- Module Selection/replication
- Communication Optimization
- Module-level Scheduling

**Inter-Module Optimization**

**Polyhedral-Based Data Reuse Optimization for Configurable Computing**
FPGA’13 Best Paper Award

**Improving Polyhedral Code Generation for High-Level Synthesis**
CODES-ISSS’14 Best Paper Award

**Theory and Algorithm for Generalized Memory Partitioning in High-Level Synthesis**
FPGA’14

**An optimal microarchitecture for stencil computation acceleration based on non-uniform partitioning of data reuse buffers**
DAC’14

**Combining Computation with Communication Optimization in System Synthesis for Streaming Applications**
FPGA’14
High-level Optimizations

Input Code (C/C++)

Loop Scheduling Optimization
- Program Analysis
- Scheduling optimization
- Code Generation

Data Layout Optimization
- Array Partitioning
- Data Reuse

Inter-Module Optimization
- Module Select./Repl.
- Communication Optimization
- Module-level Scheduling

Theory and Algorithm for Generalized Memory Partitioning in High-Level Synthesis, FPGA’14
Memory Partitioning for Throughput Optimization

- Memory is still a bottleneck
  - Data intensive applications: image/video
  - Loop unrolling/tiling/pipelining

- Memory partitioning

Size = K, Bandwidth = p
p: memory port number

Size ≈ K, Bandwidth= N*p
N: Partition Factor

Challenge: generate conflict-free memory partitioning for a given program
Cyclic partitioning
- Easy to implement
- Very effective in practice

Example
- Will \(i\) and \((3*i+1)\) go to the same memory bank?

Theorem
\[\forall i, \text{ if } a_1*i+b_1 \neq a_2*i+b_2 \mod N \]
\[\iff \text{gcd}(a_1-a_2, N) \nmid (b_1-b_2)\]

*J. Cong, W. Jiang, B. Liu and Y. Zou. ACM TODAES 2011 (Best Paper Award)
Flatten-Based Partitioning

- Flatten multidimensional array
- Partition flattened single dimensional array

```c
for (j=0; j<w1; j++)
    for (i=0; i<w0; i++)
        foo(A[j][i], A[j][i-1], A[j-1][i ], A[j+1][i], A[j][i+1]);
```

```c
foo(A[w0 *j+i], A[w0 *j+i-1], A[w0 *j+i- w0], A[w0 *j+i+ w0], A[w0 *j+i +1]);
```

Conflict free conditions:

- \( N \nmid 2 \)
- \( N \nmid w0 \)
- \( N \nmid (w0-1) \)
- \( N \nmid (w0+1) \)

Partition results are related to array sizes!
Linear-Transformation-Based Partitioning*

- Linear transformation-based approach
  - Multidimensional address $\vec{x}$ linearization: $L(\vec{x}) = \vec{\alpha} \cdot \vec{x}$
  - Bank mapping: $\text{bank}(\vec{x}) = L(\vec{x}) \mod N$ (Cyclic)
- Example: denoise

∀ $i,j$, $A[a_1*i+b_1][c_1*j+d_1]$ not conflict with $A[a_1*i+b_1][c_1*j+d_1]$
⇔ $\gcd((\alpha_1 (a_1-a_2)+\alpha_2 (c_1-c_2)), N) \nmid (\alpha_1 (b_1-b_2)+\alpha_2 (d_1-d_2))$

*Y. Wang, P. Li, P. Zhang, C. Zhang and J. Cong. DAC 2013
Access Conflict

for (i = 1; i <= n; i++)
  for (j = 1; j <= min(n, n-i+2); j++)
    foo(A[j+1][i+1], A[j][2*i]);

No Conflict

Conflict!

Array accesses
bank(x0, x1) = (x0 + x1) % 3

Iterator domain

j

n

2

1

i

1 2 n

2

1

x0

x1

n+1

2*n
**Conflict Polytope**

- **Conflict polytope** of two references is a subset of iteration domain where the two references are mapped on the same bank

\[
\begin{align*}
1 \leq i \leq n \\
1 \leq j \leq n \\
i + j \leq n + 2 \\
(i + 1) + (j + 1) \equiv (2i + j) \mod 3
\end{align*}
\]

- Insert an extra variable \( k \) to linearize

\[(i + 1) + (j + 1) = (2i + j) + 3k\]

- Fourier–Motzkin Algo. (Fourier 1826, Motzkin 1936)
  - Test the emptiness of the conflict polytope
  - Algorithm complexity: \( O(m^{2^t}) \)
    - \( m \): number of inequalities (\( m = 4 \) in the example)
    - \( t \): number of variables (\( t = 3 \) in the example)
    - independent of iteration domain size \( n \)

```java
for (i = 1; i <= n; i++)
    for (j = 1; j <= min(n, n-i+2); j++)
        foo(A[j+1][i+1], A[j][2*i+1]);
```

\[\text{bank}(x_0, x_1) = (x_0 + x_1) \% 3\]
Generalized Memory Partitioning (GMP)

- Complexity independent of
  - Sizes of iteration domain
  - Sizes of array

- Complexity related to
  - Dimensions of iteration domain and array
    - In real case $\leq 3$
  - Number of References
    - In real case $\leq 100$

- Number of memory partitioning alternatives $\leq$ number of references

Flowchart:

1. Memory Partitioning Alternatives
2. Build Conflict Polytopes for all ref pairs
3. All Polytopes Empty?
   - Y: Valid memory partitioning
   - N: Go back to Build Conflict Polytopes for all ref pairs
Experimental Results for Denoise

Gradient (II 4->1)

Rician (II 4->1)
Optimization Beyond HLS

Input Code (C/C+)

Loop Structure Optimization
- Program Analysis
- Loop Restructuring
- Code Generation

Data Layout Optimization
- Array Partitioning
- Data Reuse

Inter-Module Optimization
- Module Selection/replication
- Communication Optimization
- Module-level Scheduling

Combining Computation with Communication Optimization in System Synthesis for Streaming Applications, FPGA’14
Motivation

Tile size: 32x32
Image: 64x64, 4 tiles

Which implementation to use for each module?

- Memory partitioned v.s. Memory non-partitioned

<table>
<thead>
<tr>
<th></th>
<th>BRAM</th>
<th>DSP</th>
<th>FF</th>
<th>LUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-partitioned gradient</td>
<td>128</td>
<td>21</td>
<td>2511</td>
<td>2125</td>
</tr>
<tr>
<td>partitioned gradient</td>
<td>176</td>
<td>56</td>
<td>7147</td>
<td>7262</td>
</tr>
<tr>
<td>partitioned rician</td>
<td>128</td>
<td>22</td>
<td>4692</td>
<td>3991</td>
</tr>
<tr>
<td>non-partitioned rician</td>
<td>176</td>
<td>88</td>
<td>14475</td>
<td>15537</td>
</tr>
</tbody>
</table>
Motivation

Tile size: 32x32
Image: 64x64, 4 tiles

- How many number of replicas?
- Scheduling and Communication cost (number of tiles in the communication channel)?

scheduling 0 $\rightarrow$ 1 tile
scheduling 0 $\rightarrow$ 2 tiles
Goal of this work: understand and explore the design space of mapping streaming applications to FPGAs

What is the minimum buffer size?
- Producer/Consumer data rate matching problem

How to schedule all the modules?
Proposed System Synthesis Framework

System Throughput Requirement
SDF-based Application Modeling
Implementation Library

Coupled Computation and Communication Optimizations

Selected Implementation
Number of Replica
Buffer Size
Scheduling Results
Formulation (1/2)

- Derive a scheduling graph
  - Associate each node with a time variable, denoting the starting time of the node
  - Scheduling graph: delineates all the scheduling constraints
    - Module latency, Module replication, System throughput requirement, Buffer constraints

```
Module latency constraints
et_a: execution time of task a  
et_b: execution time of task b

Buffer Constraints
If buffer size between a and b is 2, then add edges: b^0 \rightarrow a^2 \ b^1 \rightarrow a^3
```
Formulation (2/2)

- Associate each node with a scheduling variable
  - $t(b^0) - t(a^0) \geq e_{t_a}$
  - $t(a^2) - t(b^0) \geq e_{t_b}$
  - ...
  - Scheduling variables are integer variables

- Schedulability checking problem is a **System of Difference Constraints (SDC)** problem
  - It can be solved optimally *in polynomial time* by linear programming relaxation
  - And the solution is guaranteed to be integers
Exploration

Scheduling Graph
- Find the length of longest path ($maxL$)
- In this example, $maxL = 8$

Find critical paths
- Find all the paths whose lengths are $maxL$,
- Or more aggressively, $(1-\epsilon)*maxL$

Module Improvement
- Associate each edge a new weight – the area penalty to remove this edge from the critical paths
- Find a minimum cut on the graph
Streaming Synthesis (ST-Syn)

- **Formulation** – *Schedulability checking*
  - System of Difference Constraint Problem

- **Exploration** – *Identify critical path, module/buffer improvement*
  - Find $\epsilon$-critical paths in the scheduling graph
  - Minimum cut problem

- All can be solved by linear programming relaxation

Start from the impl with the smallest logic, minimum buffer size

---

**Schedulability Checking**

**Identify $\epsilon$-Critical Paths & Module/Buffer Size Improvement**

Success

Fail

done
Experiments on Denoise

- Our methodology: ST-Syn
  - computation & communication co-optimization
- Separate:
  - separate computation opt. + communication opt.
- Communication and computation should be considered in a unified framework

Average area reduction: 47%
Levels of Customization

- **Single-chip level**
  - Require new processor designs, e.g. using composable accelerators [ISLPED’ 12, DAC’14]

- **Server node level**
  - Host CPU + FPGA via PCI-e or QPI connections

- **Data center level**
  - Clusters of heterogeneous computing nodes
  - How about programming at data center level?
FPGA “FARM” at UCLA – A Small Multi-FPGA Rack

- Deployed in 2013
- Used for research and teaching
  - CS133 (60+ students)
  - CS259 (18 students)
Datacenter Level Integration at Microsoft

A. Putnam, “A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services”, ISCA’2014
Accelerating Large-Scale Services – Bing Search

1,632 Servers with FPGAs Running Bing Page Ranking Service (~30,000 lines of C++)

A. Putnam, “A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services”, ISCA’2014
Hardware Acceleration: FPGA-Based Cluster

- A 11-node cluster with FPGA-based accelerators
  - Run on top of Spark and Hadoop (HDFS)

Each node:
1. Two Xeon processors
2. One FPGA PCIe card (Alphadata)
3. 64 GB RAM
4. 10GBE NIC

Alphadata board:
1. Virtex 7 FPGA
2. 16GB on-board RAM

master / driver

11 workers
Scalable Big-Data Programming

- **Simplified programming models**
  - MapReduce, Dataflow
- **User-transparent Runtime**
  - Distributed computing
  - Scheduling and resource management
  - Fault-tolerance

```scala
val points = sc.textfile().cache()
for (i <- 1 to ITERATIONS) {
  val gradient = points.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x)))
    - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= gradient
}
```

Spark Driver → Spark Master → Spark Worker
Spark Worker
Traditional FPGA Programming

Application FPGA Platforms

Inter-node communication:
- OpenMPI

Heterogeneous hardware:
- OpenCL (SDAccel)

Lots of setup/initiation codes
Too much hardware-specific knowledge
- Data-transfer between host and accelerator
- Manual data partition, task scheduling

Only support single application
Lack of portability

```c
int main(int argc, char** argv) {
    int npprocs = 0;
    int rank = 0;
    int namelen;
    char processor_name[ MPI_MAX_PROCESSOR_NAME ];
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &npprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(processor_name, &namelen);

    int L = LABEL_SIZE;
    int H = HIDDEN_SIZE;
    int D = FEATURE_SIZE;
    int n = 60000;
    int maxiter = 10;
    if ( rank == 0 ) {
        // read from file
        char *fname, *fname2;
        if ( argc > 2 ) {
            fname = argv[1];
            fname2 = argv[2];
        } else {
            MPI_Abort(MPI_COMM_WORLD, -1);
        }
        load_data(fname, fname2);
        float* local_dataPoints = (float*)malloc((L+D)*local_N*sizeof(float));
        MPI_Scatter(local_dataPoints, local_N*(L+D), MPI_FLOAT,
        local_dataPoints, local_N*(L+D), MPI_FLOAT, 0,
        MPI_COMM_WORLD);
        if( rank == 0 ) clear_data();
    }
    
    cl_kernel kernel;  // compute kernel
    char cl_platform_vendor[1001];
    char cl_platform_name[1001];
    cl_mem input_weights;
    cl_mem input_data;
    cl_mem output_gradient;
    err = clGetPlatformIDs(1, &platform_id, NULL);
    err = clGetPlatformInfo(platform_id, CL_PLATFORM_VENDOR, 1000,
    (void*)cl_platform_vendor, NULL);
    err = clGetPlatformInfo(platform_id, CL_PLATFORM_NAME, 1000, 
    (void*)cl_platform_name, NULL);
    
    connect_to_opencl()
    int lpga = 0;
    if defined (FPGA) {
        lpga = 1;
    }
    if defined (FPGA) {
        lpga = 1;
    }
    if err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &input_weights); 
    err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &input_data);
    cl_mem input_weights;
    cl_mem input_data;
    cl_mem output_gradient;
    err = clGetPlatformIDs(1, &platform_id, NULL);
    err = clGetPlatformInfo(platform_id, CL_PLATFORM_VENDOR, 1000,
    (void*)cl_platform_vendor, NULL);
    err = clGetPlatformInfo(platform_id, CL_PLATFORM_NAME, 1000, 
    (void*)cl_platform_name, NULL);
    
    // Set the arguments to our compute kernel
    err = 0;
    err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &input_weights);
    err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &input_data);
    
    // Call kernel
    err = clEnqueueStartKernel(kernel, 0, NULL);
    err = clEnqueueReadBuffer(commands, input_weights, CL_TRUE, 0, 
    sizeof(int) * n*(D+L), global_data, 0, NULL, NULL);
    } // if(iter==0) {
    err = clEnqueueWriteBuffer(commands, input_data, CL_TRUE, 0, 
    sizeof(int) * n*(D+L), global_data, 0, NULL, NULL);
    
    // Wait for events
    } while (iter < 10) {
        if err = clWaitForEvents(1, &readevent); 
        err = clEnqueueReadBuffer(commands, input_weights, CL_TRUE, 0, 
        sizeof(int) * weight_size, global_weights, 0, NULL, &readevent);
        
    MPI_Barrier(MPI_COMM_WORLD);
    
    for (iter=0; iter<10; ++iter++) {
        // Do some work here
    }
}
```
System Overhead for FPGA Acceleration

Example of AlphaData FPGA platform:

Spark Task → C/C++ Proc → FPGA DRAM → FPGA BRAM

Serialization & Interprocess memcpy → PCIE memcpy → Off-chip memcpy

MNIST Dataset: ~180MB
2936ms → 255ms → 0.14ms

FPGA Accelerator time: 303ms
Minimize Overheads With Task Scheduling

Task status:  □ idle  ▪ ready  ▫ finish  ● retired

Communication Manager

Task Manager

Task Queue

Retiring Queue

Executor

Spark Task

AccRDD

Spark Task

AccRDD

Spark Task

AccRDD

input data

output data
Task Pipeline Visualization

ACC Task

Naïve method

Blaze Runtime
Minimize Overheads with Data Caching

- Take advantage of iterative computation
  - Input data set shared between iterations
  - Extends Spark RDD → AccRDD

- Block Manager – manages logic cache table
  - ①: Block in CPU memory
  - ②: Block in FPGA memory
## Experimental Results

- **One iteration of Logistic Regression on MNIST**
  - 60,000 points, 784 features
  - $K^{th}$ iteration (data cached on Spark)

- **Graph:**

<table>
<thead>
<tr>
<th>Designs</th>
<th>Execution Time (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>3,119</td>
</tr>
<tr>
<td>ACC-base</td>
<td>1,880 (31.4%)</td>
</tr>
<tr>
<td>ACC-cache</td>
<td>1,375 (50.5%)</td>
</tr>
<tr>
<td>ACC-board-cache</td>
<td>1,262 (87.3%)</td>
</tr>
</tbody>
</table>

- **Diagram Labels:**
  - Driver
  - Spark task
  - ACC execution
  - PCIE Copy
  - JVM Copy
  - Communication
  - Broadcast

---

59
More on Cluster Accelerator Resource Management

How to make sure acc resources are efficiently utilized?

- Each framework get the accelerator resources it deserves
- Tasks in each framework are assigned to their preferred node
- Fine-grained sharing of each accelerator

![Diagram of Cluster Accelerator Resource Management](image)
Falcon Computing Solutions, Inc

User Applications in MapReduce/Hadoop + Java/C/C++/OpenMP

User Applications in Hadoop/Spark + Java/Scala

Overall Computing Solutions

FCS Compiler

ACC Models

ACC Libraries: FCS or 3rd party

FCS Runtime

The only solution of FPGA customization and virtualization for Datacenter acceleration!

Customize & Virtualize

ACC: accelerator

ACC Engines

©Copyright 2015 Falcon Computing Solutions
Challenges and Opportunities Above and Beyond HLS

- Enable customized computing at datacenter scale
  - Better programming environment and compilation
  - Runtime management
- Enable rapid design and deployment of IoTs
Figure Source: Mark Fell. "Roadmap for the Emerging Internet of Things - Its Impact, Architecture and Future Governance". Carré & Strauss, United Kingdom, 2014
Needs for an IoT Compiler

- Expect many kinds of IoTs, one for each application domain
  - Enable an IoT Compiler to enable “printing” of IoT chips …

- New requirements for an IoT Compiler
  - Interface with an IoT-core and IoT-OS
  - Interface with an IoT-bus/NoC
  - Interface with various sensing and communication devices (via the IoT-bus)
  - Aggressive power optimization (we didn’t get a chance to do too much in AutoESL)

- Enable rapid design of IoT-on-a-chip!
  - Expect to hear more from Ispirit!
Concluding Remarks

- High-level synthesis (HLS) is gaining wider adoption

- There are plenty of opportunities above and beyond HLS
  - Enable customized computing at datacenter scale
    - Better programming environment and compilation
    - Runtime management
  - Enable rapid design and deployment of IoTs
Acknowledgements – NSF, Intel, and C-FAR

- NSF Expeditions in Computing Program and InTrans Program with Intel
- C-FAR Center under the STARnet Program
- CDSC faculty:

- Aberle (UCLA)
- Baraniuk (Rice)
- Bui (UCLA)
- Chang (UCLA)
- Cheng (UCSB)
- Cong (Director) (UCLA)
- Palsberg (UCLA)
- Potkonjak (UCLA)
- Reinman (UCLA)
- Sadayappan (Ohio-State)
- Sarkar (Associate Dir) (Rice)
- Vese (UCLA)
Postdocs, Graduate Students, and Collaborators

Prof. Deming Chen (UIUC/ADSC)

Yuting Chen (UCLA)

Hui Huang (UCLA)

Muhuan Huang (UCLA)

Dr. Peng Li (UCLA)

Prof. Louis-Noël Pouchet (UCLA)

Yuxin Wang (PKU)

Di Wu (UCLA)

Bingjun Xiao (UCLA)

Hao Yu (UCLA)

Dr. Peng Zhang (UCLA)

Yi Zou (UCLA)

Wei Zuo (UIUC)