## Tutorial Outline

<table>
<thead>
<tr>
<th>Time</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>8:30 am – 9:00 am</td>
<td>Introduction</td>
</tr>
<tr>
<td>9:00 am – 10:00 am</td>
<td><strong>Pre-RTL Simulation Framework: Aladdin</strong></td>
</tr>
<tr>
<td>10:00 am – 10:30 am</td>
<td>Break</td>
</tr>
<tr>
<td>10:30 am – 11:00 am</td>
<td>Workload Characterization Tool: WIICA</td>
</tr>
<tr>
<td>11:00 am – 12:00 pm</td>
<td>CAD &amp; Benchmarks: HLS &amp; MachSuite</td>
</tr>
<tr>
<td>12:00 pm – 2:00 pm</td>
<td>Lunch</td>
</tr>
<tr>
<td>2:00 pm – 3:00 pm</td>
<td>Embedded Keynote Talk: Mark Horowitz (Stanford)</td>
</tr>
<tr>
<td>3:00 pm – 3:30 pm</td>
<td>Accelerator Selection Tool: Sigil</td>
</tr>
<tr>
<td>3:30 pm – 4:00 pm</td>
<td>Break</td>
</tr>
<tr>
<td>4:00 pm – 5:00 pm</td>
<td>Hands-on Exercise</td>
</tr>
</tbody>
</table>
ALADDIN
A Pre-RTL, Power-Performance Accelerator Simulator
Enabling Large Design Space Exploration of Customized Architectures

Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, David Brooks
Harvard University
Today’s SoC
Future Accelerator-Centric Architectures

- How to decompose an application to accelerators?
- How to rapidly design lots of accelerators?
- How to design and manage the shared resources?

Flexibility
Design Cost
Programmability
Aladdin: A pre-RTL, Power-Performance Accelerator Simulator

Unmodified C-Code → Aladdin

Accelerator Design Specific Datapath
Parameters (e.g., # FU, mem. BW) → Aladdin

Private L1/Scratchpad → Performance

Shared Memory/Interconnect Models → Power/Area

“Accelerator Simulator”
Design Accelerator-Rich SoC Fabrics and Memory Systems

FP

(1000, 1000)

(1000, 1000)
Aladdin: A pre-RTL, Power-Performance Accelerator Simulator

“Accelerator Simulator”
Design Accelerator-Rich SoC Fabrics and Memory Systems

Flexibility
Programmability

Unmodified C-Code
Accelerator Design Specific Parameters (e.g., # FU, mem. BW)

Shared Memory/Interconnect Models

Power/Area
Performance
Aladdin: A pre-RTL, Power-Performance Accelerator Simulator

“Accelerator Simulator”
Design Accelerator-Rich SoC Fabrics and Memory Systems

Unmodified C-Code
Accelerator Design Parameters (e.g., # FU, mem. BW)

Aladdin
Accelerator Specific Datapath
Private L1/Scratchpad

Shared Memory/Interconnect Models

Power/Area
Performance

“Design Assistant”
Understand Algorithmic-HW Design Space before RTL

Flexibility
Programmability

Design Cost
Future Accelerator-Centric Architecture

- Big Cores
- Small Cores
- Shared Resources
- Memory Interface
- Sea of Fine-Grained Accelerators

Graph:
- X-axis: Execution Time (μS)
- Y-axis: Power (mW)
- HLS
**Future Accelerator-Centric Architecture**

- **GPU/DSP**
  - Big Cores
  - Small Cores
  - Shared Resources
  - Memory Interface
  - Sea of Fine-Grained Accelerators

---

Aladdin can **rapidly** evaluate large design space of accelerator-centric architectures.
Aladdin Overview

C Code

Optimization Phase

Dynamic Data Dependence Graph (DDDG)

Realization Phase

Power/Area

Optimistic IR

Initial DDDG

Idealistic DDDG

Program Constrained DDDG

Resource Constrained DDDG

Power/Area Models

Performance

Activity

Acc Design Parameters
Aladdin Overview

Optimization Phase

C Code

Optimistic IR → Initial DDDG → Idealistic DDDG

Acc Design Parameters

Program Constrained DDDG → Resource Constrained DDDG → Power/Area Models

Performance

Activity

Power/Area

Realization Phase
Aladdin is NOT

• An HLS flow:
  – No RTL is generated.
  – High-level estimates of power and performance;
  – Aladdin uses fully dynamic analysis to expose algorithmic parallelism for unmodified HLL codes;

• Limit of ILP study:
  – “optimistic but realistic” DDDG is constructed to model accelerators.
From C to Design Space

C Code:
for(i=0; i<N; ++i)
c[i] = a[i] + b[i];
Aladdin Overview

Optimization Phase

C Code

Optimistic IR

Initial DDDG

Idealistic DDDG

Program Constrained DDDG

Resource Constrained DDDG

Power/Area Models

Realization Phase

Performance

Activity

Power/Area

Acc Design Parameters
From C to Design Space

IR Dynamic Trace

C Code:

```
for(i=0; i<N; ++i)
c[i] = a[i] + b[i];
```

<table>
<thead>
<tr>
<th>Step</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(r_0 = 0) // i = 0</td>
</tr>
<tr>
<td>1</td>
<td>(r_4 = \text{load } (r_0 + r_1)) // load( a[i] )</td>
</tr>
<tr>
<td>2</td>
<td>(r_5 = \text{load } (r_0 + r_2)) // load( b[i] )</td>
</tr>
<tr>
<td>3</td>
<td>(r_6 = r_4 + r_5)</td>
</tr>
<tr>
<td>4</td>
<td>(\text{store } (r_0 + r_3, r_6)) // store( c[i] )</td>
</tr>
<tr>
<td>5</td>
<td>(r_0 = r_0 + 1) // ++i</td>
</tr>
<tr>
<td>6</td>
<td>(r_4 = \text{load } (r_0 + r_1)) // load( a[i] )</td>
</tr>
<tr>
<td>7</td>
<td>(r_5 = \text{load } (r_0 + r_2)) // load( b[i] )</td>
</tr>
<tr>
<td>8</td>
<td>(r_6 = r_4 + r_5)</td>
</tr>
<tr>
<td>9</td>
<td>(\text{store } (r_0 + r_3, r_6)) // store( c[i] )</td>
</tr>
<tr>
<td>10</td>
<td>(r_0 = r_0 + 1) // ++i</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Optimistic IR

- LLVM IR
- High-level IR:
  - Machine- and ISA-independent
- Features:
  - Unlimited Registers
  - Simple Opcodes: add, mul, sin, sqrt
  - Only load/store access memory

Shao, et al., ISA-Independent Workload Characterization and Implications for Specialized Architecture, ISPASS, 2013
Aladdin Overview

Optimization Phase

C Code

Optimistic IR → Initial DDDG → Idealistic DDDG

Acc Design Parameters

Program Constrained DDDG → Resource Constrained DDDG → Power/Area Models

Performance

Activity

Power/Area

Realization Phase
C Code:
for(i=0; i<N; ++i)
c[i] = a[i] + b[i];

IR Trace:
0. r0=0 //i = 0
1. r4=load (r0 + r1) //load a[i]
2. r5=load (r0 + r2) //load b[i]
3. r6=r4 + r5
4. store(r0 + r3, r6) //store c[i]
5. r0=r0 + 1 //++i
6. r4=load(r0 + r1) //load a[i]
7. r5=load(r0 + r2) //load b[i]
8. r6=r4 + r5
9. store(r0 + r3, r6) //store c[i]
10. r0 = r0 + 1 //++i
...
Aladdin Overview

Optimization Phase

C Code → Optimistic IR → Initial DDDG → Idealistic DDDG

Realization Phase

Acc Design Parameters → Program Constrained DDDG → Resource Constrained DDDG → Power/Area Models

Performance → Activity → Power/Area
C Code:
for(i=0; i<N; i++)
c[i] = a[i] + b[i];

IR Trace:
0. r0=0  //i = 0
1. r4=load (r0 + r1) //load a[i]
2. r5=load (r0 + r2) //load b[i]
3. r6=r4 + r5
4. store(r0 + r3, r6) //store c[i]
5. r0=r0 + 1  //++i
6. r4=load(r0 + r1) //load a[i]
7. r5=load(r0 + r2) //load b[i]
8. r6=r4 + r5
9. store(r0 + r3, r6) //store c[i]
10. r0 = r0 + 1  //++i
...

From C to Design Space
Idealistic DDDG
• Include application-specific customization strategies.
• Node-Level:
  – Bit-width Analysis
  – Strength Reduction
  – Tree-height Reduction
• Loop-Level:
  – Remove dependences between loop index variables
• Memory Optimization:
  – Memory-to-Register Conversion
  – Store-Load Forwarding
  – Store Buffer
• Extensible
  – e.g. Model CAM accelerator by matching nodes in DDDG
Aladdin Overview

Optimization Phase

Optimistic IR → Initial DDDG → Idealistic DDDG

Realization Phase

C Code

Acc Design Parameters

Performance

Activity

Power/Area

Program Constrained DDDG → Resource Constrained DDDG → Power/Area Models
From C to Design Space

One Design

Idealistic DDDG

Acc Design Parameters:
✓ Memory BW <= 2
✓ 1 Adder

Resource Activity

Cycle
From C to Design Space

Another Design

Idealistic DDDG

Resource Activity

Acc Design Parameters:
✓ Memory BW <= 4
✓ 2 Adders

Cycle
From C to Design Space

Realization Phase: DDDG->Power-Perf

• Constrain the DDDG with program and user-defined resource constraints

• Program Constraints
  – Control Dependence
  – Memory Ambiguation

• Resource Constraints
  – Loop-level Parallelism
  – Loop Pipelining
  – Memory Ports
  – # of FUs (e.g., adders, multipliers)
Memory Ambiguation

• Idealistic DDDG optimistically removes *all* false memory dependences
• Input-dependent memory accesses cannot be calculated statically.
for(i=0; i<N; ++i) {
    bucket[ a[i] & 0x11 ]++;
}

Input:

a[0] = 1;

Memory Ambiguation

0.i=0
1.ld a[0]
2.&
3.ld b[1]
4.b[1]++
5.st b[1]
for(i=0; i<N; ++i)
{
    bucket[ a[i] & 0x11 ]++;
}

Input:

a[0] = 1;
a[1] = 2;
for(i=0; i<N; ++i) {
    bucket[a[i] & 0x11]++;  
}

Input:

a[0] = 1;
a[1] = 2;
a[2] = 2;
...
Memory Ambiguation

for(i=0; i<N; ++i)
{
    bucket[ a[i] & 0x11 ]++; 
}

Input:

a[0] = 1;

a[1] = 2;

a[2] = 2;

...
for(i=0; i<N; ++i) {
    bucket[a[i] & 0x11]++;
}

Input:

a[0] = 1;
a[1] = 2;
a[2] = 2;
...
for(i=0; i<N; ++i) {
  bucket[a[i] & 0x11]++;
}

Input:

a[0] = 1;
a[1] = 2;
a[2] = 2;
...
for(i=0; i<N; ++i)
{
    bucket[ a[i] & 0x11 ]++;
}

Input:
  a[0] = 1;
  a[1] = 2;
  a[2] = 2;
  ...
From C to Design Space

Power-Performance per Design

Acc Design Parameters:
✓ Memory BW <= 4
✓ 2 Adders

Acc Design Parameters:
✓ Memory BW <= 2
✓ 1 Adder
From C to Design Space

Design Space of an Algorithm

![Graph showing the relationship between cycle and power.](image)
Cycle-Level Activity

![Graph showing number of active functional units and memory bandwidth over time. The graph plots the number of active functional units and memory bandwidth against time (cycles) from 0 to 800 cycles. There are peaks for FFT8, Twiddle, and Shuffle operations. The graph also includes a legend indicating the active functional units and memory bandwidth.]
Power Model

• Functional Units Power Model
  – Microbenchmarks characterize various FUs.
  – Design Compiler with 40nm Standard Cell

\[
Power = \sum_{1<i<N} (activity_i \times P_{i\text{dynamic}}) + P_{i\text{leakage}}
\]

• SRAM Power Model
  – Commercial register file and SRAM memory compilers with the same 40nm standard cell library
Aladdin Overview

C Code
Optimistic IR -> Initial DDDG -> Idealistic DDDG

Acc Design Parameters
Program Constrained DDDG -> Resource Constrained DDDG -> Power/Area Models

Optimization Phase
Realization Phase

Performance
Activity
Power/Area
Aladdin Validation

C Code → Aladdin → Power/Area, Performance → Design Compiler, ModelSim → Activity → Verilog → Aladdin → C Code

39
Aladdin Validation

C Code

RTL Designer

HLS C Tuning

Vivado HLS

Verilog

Design Compiler

ModelSim

Power/Area

Performance

Aladdin
## Validation Benchmarks

<table>
<thead>
<tr>
<th>Type</th>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SHOC Benchmark Suite</strong></td>
<td>MD</td>
<td>Pairwise calculation of the L-J Potential</td>
</tr>
<tr>
<td></td>
<td>STENCIL</td>
<td>Apply 3x3 filter to an image</td>
</tr>
<tr>
<td></td>
<td>FFT</td>
<td>1D 512 FFT</td>
</tr>
<tr>
<td></td>
<td>GEMM</td>
<td>Blocked Matrix Multiply</td>
</tr>
<tr>
<td></td>
<td>TRIAD</td>
<td>Single Computation in DOALL loop</td>
</tr>
<tr>
<td></td>
<td>SORT</td>
<td>Radix Sort</td>
</tr>
<tr>
<td></td>
<td>SCAN</td>
<td>Parallel prefix sum</td>
</tr>
<tr>
<td></td>
<td>REDUCTION</td>
<td>Return sum of an array</td>
</tr>
<tr>
<td><strong>Proposed Accelerator Constructs</strong></td>
<td>NPU</td>
<td>An individual neuron in a network [MICRO'12]</td>
</tr>
<tr>
<td></td>
<td>Memcached</td>
<td>GET function in Memcached [ISCA'13]</td>
</tr>
<tr>
<td></td>
<td>HARP</td>
<td>Data partition accelerator [ISCA'13]</td>
</tr>
</tbody>
</table>

Optimized HLS Designs

Hand RTL Designs
Aladdin Validation

- Time (KCycles)
  - 0.6%

- Power (mW)
  - 6.5%

- Area (mm²)
  - 4.5%

Graphs comparing different scenarios and flows.
Aladdin Validation

- Time (KCycles) for various operations:
  - 0.9% improvement for NPU
  - 4.9% improvement for HASH
  - 6.5% improvement for HARP

- Power (mW) for various operations:

- Area (mm²) for various operations:
Aladdin enables rapid design space exploration for accelerators.
Limitations

• Algorithm Choices
  – Aladdin generates a design space per algorithm
  – Can use Aladdin to quickly compare the design spaces of algorithms

• Input Dependent
  – Inputs that exercise all paths of the code

• Input C Code
  – Aladdin can create DDDG for any C code.
  – C constructs that require resources outside the accelerator, such as system calls and dynamic memory allocation, are not modeled.
Aladdin enables pre-RTL simulation of accelerators with the rest of the SoC.
Simulating Accelerator with Memory System using Aladdin

![Diagram showing power consumption over time with L1 and L2 sizes and bandwidths as variables.]

- L1 Size
- L2 Size
- L1 BW
- L2 BW
- Acc
- Cache
- Memory

- Power (mW) vs Time (Million Cycles)
- 256 Bytes/Cycle
- 128 Bytes/Cycle
- 64 Bytes/Cycle
- 32 Bytes/Cycle
- 16 Bytes/Cycle
- 8 Bytes/Cycle
- 4 Bytes/Cycle
Modeling Accelerators in an SoC-like Environment
Modeling Accelerators in a SoC-like Environment

![Diagram showing power consumption over time for different memory block sizes with and without memory contention.](image)
Architectures with 1000s of accelerators will be radically different; New design tools are needed.

Aladdin enables rapid design space exploration of future accelerator-centric platforms.

You can find Aladdin at
  – http://vlsiarch.eecs.harvard.edu/accelerators
Tutorial References


- http://vlsiarch.eecs.harvard.edu/accelerators