High-Level Synthesis

CUSO Winter School 2020
Champéry

Paolo Ienne
paolo.ienne@epfl.ch
Moore’s Law and Other Exponentials...

40 Years of Microprocessor Trend Data

- Transistors (thousands)
- Single-Thread Performance (SpecINT x 10^3)
- Frequency (MHz)
- Typical Power (Watts)
- Number of Logical Cores
Computing = Processors (CPUs, GPUs,...)

Take a standard circuit (a processor)

Use its resources over and over in time

What to do is in program memory and can be changed

It is AWFULLY SLOW
Or NotNecessarily?! 

Temporal Computing

R1: A
R2: B
R3: C
R4: X
R5: tmp
R6: Y

ALU

Spatial Computing

MUL R4 R4 R5
MUL R4 R2 R6
ADD R3 R6 R6
MUL R1 R5 R5
MUL R5 R6 R6

MUL R4 R4 R5
MUL R4 R2 R6
ADD R3 R6 R6
MUL R1 R5 R5
MUL R5 R6 R6

Adapted from DeHon, © ACM 1999
Spatial Computing

FAST and EFFICIENT computing paradigm

Does it have only a single fixed function, now?!

How to design the appropriate circuit for a given application?!
Field Programmable Gate Arrays

**Programmable** circuits:

- Millions of **Logic Blocks** (i.e., look-up tables, that is arbitrary logic functions, and flip-flops)
- Thousands of **Block RAMs** (i.e., small memories)
- Thousands of **DSP Units** (i.e., simple ALUs)
- Gazillions of configurable switches to interconnect everything

https://medium.com/@ckyrkou
Field Programmable Gate Arrays

Worth keeping in mind:

- A look-up table is much slower than a gate
- Selecting things (different paths, components, flip-flops) costs area and time
- Connecting look-up tables now involves many transistors, not just metal wires

FPGAs are slow and expensive!
(compared to how processors are built)
Spatial Computing

FAST and EFFICIENT computing paradigm

FPGA make a programmable function possible

How to design the appropriate circuit for a given application?!
How do we *program* FPGAs?
From Programs...

```c
#define PI 3.1415926535897932384626434

complex* DFT_naive(complex* x, int N) {
    complex* X = (complex*) malloc(sizeof(struct complex_t) * N);
    int k, n;
    for (k = 0; k < N; k++) {
        X[k].re = 0.0;
        X[k].im = 0.0;
    }
    for (n = 0; n < N; n++) {
        X[k] = add(X[k], multiply(x[n],
                     conv_from_polar(1,
                     -2*PI*n*k/N)));
    }
    return X;
}
```
#define PI 3.1415926535897932384626434

complex* DFT_naive(complex* x, int N) {
    complex* X = (complex*) malloc(sizeof(struct complex_t) * N);
    int k, n;
    for(k = 0; k < N; k++) {
        X[k].re = 0.0;
        X[k].im = 0.0;
        for(n = 0; n < N; n++) {
            X[k] = add(X[k], multiply(X[n],
                                  conv_from_polar(1,
                                                  -2*PI*n*k/N)));
        }
    }
    return X;
}
A Bit of History

- **Generation 0 (1970s), prehistory**
  - Groundbreaking academic work

- **Generation 1 (1980s until early 1990s)**
  - Mostly important academic work; few commercial players
  - Focus on scheduling, binding, etc.
  - Almost competing in adoption with RTL logic synthesis

- **Generation 2 (mid 1990s until early 2000s)**
  - Main EDA players offer commercial HLS tools; commercial failure
  - Assumed RTL designers would embrace the technology, but there was not enough gain for them
  - Wrong programming languages (VHDL or new languages)

- **Generation 3 (from early 2000s)**
  - Currently available commercially (e.g., Vivado HLS); some successes
  - Connected to the rise of FPGAs (fast turnaround, inexperienced designers, etc.)
  - Focus on C/C++ and on demanding dataflow/DSP applications
  - Better results (progress in compilers, including VLIW)
Outline

1. Moore, spatial computing, FPGAs, HLS

2. Why hardware?
   Why do we expect hardware to be better than software...

3. HLS #1: Instruction set extensions
   Not really what most people see as HLS

4. HLS #2: Classic statically scheduled HLS
   The most common form of HLS today

5. HLS #3: Dataflow circuits
   The out-of-order superscalar processor equivalent in HLS

6. Some caveats in form of conclusions
Processors

multicycle processor

pipelined processor
Elementary Motivational Example
An Important Kernel...

/* init */
a <<= 8;
/* loop */
for (i = 0; i < 8; i++) {
    if (a & 0x8000) {
        a = (a << 1) + b;
    } else {
        a <<= 1;
    }
}
return a & 0xffff;

Shift-and-add
unsigned
8 x 8-bit
multiplication
Software Predication

/* init */
a <<= 8;

/* loop */
for (i = 0; i < 8; i++) {
    p1 = - ((a & 0x8000) >> 15);
    a = (a << 1) + b & p1;
}
return a & 0xffff;

Predicate mask (0 or -1 = 0xffffffff)

Shift  Predicated Add
Loop Kernel DAG

In SW

In HW

~6 cycles

1-2 cycles!

Only wiring

AND gates

ALU
New Unit To Accelerate Shift-and-Add Multiplication Loop

One instruction added

⇒

loop kernel reduced to **15-30%**
Loop Unrolling

/* init */
a <<= 8;
/* no loop anymore */

p1 = - ((a & 0x8000) >> 15); a = (a << 1) + b & p1;
p1 = - ((a & 0x8000) >> 15); a = (a << 1) + b & p1;
p1 = - ((a & 0x8000) >> 15); a = (a << 1) + b & p1;
p1 = - ((a & 0x8000) >> 15); a = (a << 1) + b & p1;
p1 = - ((a & 0x8000) >> 15); a = (a << 1) + b & p1;
p1 = - ((a & 0x8000) >> 15); a = (a << 1) + b & p1;
p1 = - ((a & 0x8000) >> 15); a = (a << 1) + b & p1;

return a & 0xffff;
Full DAG

In SW

- \( a \times 8000 \)
- \( 1 \& 15 \)
- \( \gg 1 \)
- \( 0x8000 \)
- \( + \)
- \( \& 15 \)
- \( \gg 1 \)
- \( \& 1 \)
- \( \gg 8 \)
- \( + \)
- \( \& 1 \)
- \( \gg 8 \)
- \( + \)
- \( \& 1 \)
- \( \gg 8 \)
- \( + \)
- \( \& 1 \)
- \( \gg 8 \)
- \( + \)
- \( \& 1 \)
- \( \gg 8 \)
- \( + \)
- \( \& 1 \)
- \( \gg 8 \)

\(~50\) cycles

In HW

- \( a \)
- \( b \)
- \( \& \)
- \( \&-network \)
- \( Column\ Compr. \)
- \( + \)
- \( a \)

\(~3-4\) cycles

Arithmetic Optimiser

Etc.
New Unit To Accelerate Multiplication?! Yeah, a MUL...

Rn ← (Rn & 0x0000.ffff) x (Rm & 0x0000.ffff)

One instruction added
⇒
function reduced by a factor 10-15
If I am in a context where I can implement whichever processor I want (e.g., on an FPGA)...

...then I can adapt the processor to the applications and needs I have.

**Classic “Specialisation”...**
Why Hardware Is Better?

• Spatial computation
  – Cheap “ILP” without true ILP support

• No quantization of time in clock cycles for each operation/instruction
  – Operation chaining

• Hardware is a different
  – Constants may be propagated
  – Precision can be tuned (bitwidth analysis)
  – Arithmetic components can be optimized
Spatial Computation

Spatial Computing

Temporal Computing

Parallelism

Adapted from DeHon, © ACM 1999
Spatial Computation

Spatial Computing

Temporal Computing

Chaining?!
No Time Quantization

- Effective occupation of the execute stage

![Diagram showing the comparison between w/o AFU and with AFU in terms of effective occupation of the execute stage.](image)
/* an excerpt from adpcm.c */
/* adpcmdecoder, mediabench */

vpdiff = step>>3;
if (delta & 4) vpdiff += step;
if (delta & 2) vpdiff += step>>1;
if (delta & 1) vpdiff += step>>2;

- Exploited to reduce complexity
  
  \[ a \times 5 \rightarrow a << 2 + a \]

- Hardcoded into logic

- Bitwise operations (e.g., on `delta`, `step`) reduce to wires
Bitwidth Analysis
Example

/* an excerpt from adpcm.c */
/* adpcmencoder, mediabench */

index = indexTable[delta];

if (index < 0) index = 0;
if (index > 88) index = 88;

step = stepsizeTable[index];

- $0 \leq \text{index} \leq 88$
- 7 bits sufficient for representation
- Faster arithmetic components, etc.
Arithmetic Optimizations

• Arithmetic operations often appear in groups (dataflow graphs)
• A literal/sequential implementation may not make the best of the potential available
• A different number representation can be a game-changer
  – May bring large advantages, often without higher hardware cost
  – Big O complexity $O()$ may change with a different representation!
  – E.g., carry-save adders, column compressors, etc.
• Simple example: Multiply-Accumulate ($MAC = MUL + ADD$)
  – Only marginally slower than corresponding MUL
  – Practically same complexity
Why Hardware Is Better?

- Exploit constant for logic simplification
- Some operations reduce to wires in hardware
- Exploit data parallelism in hardware
- Exploit arithmetic properties for efficient chaining of arithmetic operations (e.g., carry save)
Outline

1. Moore, spatial computing, FPGAs, HLS
2. Why hardware?
   Why do we expect hardware to be better than software...
3. HLS #1: Instruction set extensions
   Not really what most people see as HLS
4. HLS #2: Classic statically scheduled HLS
   The most common form of HLS today
5. HLS #3: Dataflow circuits
   The out-of-order superscalar processor equivalent in HLS
6. Some caveats in form of conclusions
Instruction Set Extensions (ISE)

- Collapse a subset of the Direct Acyclic Graph nodes into a single Functional Unit (AFU)
  - Exploit cheaply the parallelism within the basic block
  - Simplify operations with constant operands
  - Optimise sequences of instructions (logic, arithmetic, etc.)
  - Exploit limited precision
Formulate it as an optimization problem

Find subgraphs

1. having a user-defined maximum number of inputs and outputs,
2. convex,
3. possibly including disconnected components, and
4. that maximise the overall speedup
Automatic ISE Discovery
Automatic ISE Discovery
Examples
Processor Customization?

• Arguably the **most widespread method of designing embedded hardware**: selecting one of very many existing processors or configuring the parameters of a family of processors amounts to customization for a set of applications

• **Little automation**, though: still mostly a manual design-space exploration; glimpses of automation in the 2000s seem lost

• Automatic ISE discovery could be a more promising automatic customization opportunity, but also disappeared in the late 2000s (the “fourth generation HLS” is dead?)
  
  – **Pros**: Focus on automatic design of datapath and leave control to manually optimized processors (prediction, speculation, etc.)
  
  – **Cons**: Limited scope of exploitable parallelism (datapath parallelism and convertible control—e.g., predication, unrolling)
Outline

1. Moore, spatial computing, FPGAs, HLS
2. Why hardware?
   Why do we expect hardware to be better than software...
3. HLS #1: Instruction set extensions
   Not really what most people see as HLS
4. HLS #2: Classic statically scheduled HLS
   The most common form of HLS today
5. HLS #3: Dataflow circuits
   The out-of-order superscalar processor equivalent in HLS
6. Some caveats in form of conclusions
Beyond Dataflow

• Somehow, ISE is confined to dataflow or convertible control flow, and this limits exploitable parallelism

• Traditional **HLS gets rid of the processor** altogether

• It represents an attempt (started in the late ’80s and early ’90s) to **raise the abstraction level of hardware design** above the classic RTL level (i.e., synthesizable VHDL and Verilog)
What Circuits
Do We Want HLS to Generate?

• Output of **HLS is ill-defined**
  – An example could be to generate always the same hardware (the RTL of a software processor) and binary code for it—hardly what we usually mean by HLS...

• The informal expectation is a circuit **much more massively parallel** than what a classic software processor can achieve
What One Expects from HLS?

- HLS analyzes and **exploits the concurrency** in an algorithm
- HLS inserts registers as necessary to limit critical paths and achieve a desired clock frequency
- HLS generates control logic that directs the data path
- HLS implements interfaces to connect to the rest of the system
- HLS maps data onto storage elements to balance resource usage and bandwidth
- HLS maps computation onto logic elements performing user specified and **automatic optimizations to achieve the most efficient implementation**
Architectural Template

- We need to choose a template which we customize to and optimize for the code at hand
- Usually something of this sort:
Scheduling the Datapath

- Assign operations to functional units respecting data dependencies and functional unit latencies

ASAP, unconstrained

ASAP, constraint: 1 mul

Clock Period

Cycles
Very Long Instruction Word (VLIW) Processors

Static Scheduling:
What each unit does in each cycle is decided at compile time in software

Instruction Memory
128-512 bits
# Traditional Code vs. VLIW Code

<table>
<thead>
<tr>
<th>Traditional</th>
<th>VLIW</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000: op 1</td>
<td>1000: op 1</td>
</tr>
<tr>
<td>1001: op 2</td>
<td>1001: NOP</td>
</tr>
<tr>
<td>1002: op 3</td>
<td>1002: NOP</td>
</tr>
<tr>
<td>1003: op 4</td>
<td>1003: NOP</td>
</tr>
<tr>
<td>1004: op 5</td>
<td>1004: NOP</td>
</tr>
<tr>
<td>1005: op 6</td>
<td>1005: NOP</td>
</tr>
</tbody>
</table>

- **cycles != instructions**
- **cycles = instructions**
- **latency-independent semantics** (Unit-Assumed Latency)
- **latency-dependent semantics** (Non Unit-Assumed Latency)
Scheduling

Traditional

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000:</td>
<td>op 1</td>
</tr>
<tr>
<td>1001:</td>
<td>op 2</td>
</tr>
<tr>
<td>1002:</td>
<td>op 3</td>
</tr>
<tr>
<td>1003:</td>
<td>op 4</td>
</tr>
<tr>
<td>1004:</td>
<td>op 5</td>
</tr>
<tr>
<td>1005:</td>
<td>op 6</td>
</tr>
</tbody>
</table>

VLIW

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000:</td>
<td>op 1</td>
</tr>
<tr>
<td>1001:</td>
<td>NOP</td>
</tr>
<tr>
<td>1002:</td>
<td>NOP</td>
</tr>
<tr>
<td>1003:</td>
<td>NOP</td>
</tr>
<tr>
<td>1004:</td>
<td>NOP</td>
</tr>
<tr>
<td>1005:</td>
<td>NOP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001:</td>
<td>NOP</td>
</tr>
<tr>
<td>1002:</td>
<td>op 2</td>
</tr>
<tr>
<td>1003:</td>
<td>NOP</td>
</tr>
<tr>
<td>1004:</td>
<td>NOP</td>
</tr>
<tr>
<td>1005:</td>
<td>NOP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1002:</td>
<td>NOP</td>
</tr>
<tr>
<td>1003:</td>
<td>op 5</td>
</tr>
<tr>
<td>1004:</td>
<td>NOP</td>
</tr>
<tr>
<td>1005:</td>
<td>NOP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1003:</td>
<td>NOP</td>
</tr>
<tr>
<td>1004:</td>
<td>NOP</td>
</tr>
<tr>
<td>1005:</td>
<td>op 8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1004:</td>
<td>NOP</td>
</tr>
<tr>
<td>1005:</td>
<td>op 17</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1005:</td>
<td>op 16</td>
</tr>
</tbody>
</table>

- cycles != instructions
- latency-independent semantics
  (Unit-Assumed Latency)
- cycles = instructions
- latency-dependent semantics
  (Non Unit-Assumed Latency)
Scheduling the Datapath

• Assign operations to functional units respecting data dependencies and functional unit latencies

![Diagram of ASAP scheduling with and without constraints]
Scheduling the Datapath

- Assign operations to functional units respecting data dependencies and functional unit latencies

ASAP, unconstrained

ASAP, constraint: 1 mul
Scheduling the Datapath

- Assign operations to functional units respecting data dependencies and functional unit latencies
Same as VLIW Scheduling?

- Very similar problem but with some notable differences:
  - Exact resources are not fixed; maybe there is a constraint on their total cost (e.g., area)
  - Clock cycle may be constrained but is in general not fixed; pipelining is not fixed (e.g., combinational operations can be chained)
  - No register file (which allows connecting everything to everything) but ad-hoc connectivity (variable cost and variable time impact)
Area Optimizations

- There may be cheaper ways to achieve the best latency or trade-offs area/latency
- New problem without immediate analogy in VLIWs
Chaining and Pipelining

- Combinational operators can be chained and clock period can often be adjusted (shortest not necessarily fastest)
- Also, a new problem without immediate analogy in VLIWs

Before operation chaining and with fast clock

- Total time: $4 \times 1t = 4t$

After operation chaining and with slower clock

- Total time: $2 \times 1.4t = 2.8t$
Scheduling under Resource Constraints

• Main focus of research in the early days
• The state of the art is based on the paper by Cong & Zhang, DAC 2006:
  – Given
    • A CDFG (i.e., a program)
    • A set of constraints including dependency constraints, resource constraints, latency constraints, cycle-time constraints, and relative timing constraints
  – Construct a valid schedule with minimal latency

• Used in recent tools such as Xilinx Vivado HLS
• But... is this all we need?
Example: FIR

acc = 0;
for (i = 3; i >= 0; i--) {
    if (i == 0) {
        shift_reg[0] = x;
        acc += x * c[0];
    } else {
        shift_reg[i] = shift_reg[i-1];
        acc += shift_reg[i] * c[i];
    }
}
y = acc;

\[
y_k = \sum_{i=0}^{3} c_i x_{k-i}
\]

- The array `shift_reg` is static and represents the last 4 samples of \(x\).
- This could be in a function which receives a stream of \(x\) (the input signal) and produces at each call an element of \(y\) (the output signal).
A Literal Translation...

```c
acc = 0;
for (i = 3; i >= 0; i--) {
    if (i == 0) {
        shift_reg[0] = x;
        acc += x * c[0];
    } else {
        shift_reg[i] = shift_reg[i-1];
        acc += shift_reg[i] * c[i];
    }
}
y = acc;
```

1. If-convert control flow whenever possible
2. Implement all existing registers
3. Implement datapath for all BBs
4. Create steering wires and muxes to connect everything
Naïve FIR

```c
acc = 0;
for (i = 3; i >= 0; i--) {
    if (i == 0) {
        shift_reg[0] = x;
        acc += x * c[0];
    } else {
        shift_reg[i] = shift_reg[i-1];
        acc += shift_reg[i] * c[i];
    }
}
y = acc;
```
Manual Code Refactoring

• Direct results are very often highly suboptimal
  – See FIR example

• Users **should have a sense of what circuit they want to produce** and suggest it to HLS tools by restructuring the code
  – See coming slides

• HLS tools today are **not** really meant to **abstract away hardware design issues** from software programmers; in practice, they are more like productivity tools to help hardware designers explore quickly the space of hardware designs they may wish to produce
Naïve FIR

- We are always computing both sides of the control decision, but which one is needed in a particular iteration is perfectly evident
The loop is doing two tasks completely independent from each other (shifting the signal samples and computing the new output sample), so shall we split it into two loops?

```c
acc = 0;
for (i = 3; i > 0; i--) {
    shift_reg[i] = shift_reg[i-1];
    acc += shift_reg[i] * c[i];
}
shift_reg[0] = x;
acc += x * c[0];
y = acc;
```
Loop Fission

- Not terribly useful per se, just two independent and parallel machines
- Does this create an opportunity to unroll loop 1? Note that it contains no computation...

```c
for (i = 3; i > 0; i--) {
    shift_reg[i] = shift_reg[i-1];
}
shift_reg[0] = x;

acc = 0;
for (i = 3; i >= 0; i--) {
    acc += shift_reg[i] * c[i];
}
y = acc;
```
Loop Unrolling (loop 1)

- Loop 1 has become a “pipeline” (although a fairly degenerate one) by unrolling—this is certainly desirable regardless.
- Loop 2 is not pipelined: the initiation interval is exactly equal to the latency of the kernel—unroll?

```c
shift_reg[2] = shift_reg[1];
shift_reg[1] = shift_reg[0];
shift_reg[0] = x;

acc = 0;
for (i = 10; i >= 0; i--) {
    acc += shift_reg[i] * c[i];
}
y = acc;
```
Loop Unrolling (loop 2)

- De facto, a new iteration now starts every cycle
- But resources may be too much—and partial unrolling would achieve some pipelining but yet it would still fill and drain the pipeline every iteration

```plaintext
shift_reg[2] = shift_reg[1];
shift_reg[1] = shift_reg[0];
shift_reg[0] = x;

acc = shift_reg[3] * c[3];
acc += shift_reg[1] * c[1];
acc += shift_reg[0] * c[0];

y = acc;
```
• Perfect pipelining cannot be achieved easily by rewriting the code
• We need to schedule differently the operations within a loop so that operations of different iterations take place simultaneously
• Essentially what “software pipelining” does in VLIW compilers: now we need it so that a software program represents a hardware pipeline
• HLS needs to implement some form of modulo scheduling
Software Pipelining

• Consider the following simple C code snippet:

```c
for (i=0, i<7, i++) {
    c[i] = a[i]+1;
}
```

• The three corresponding instructions are dependent, they cannot be executed in parallel

• Goal: **restructure the loop**, so that some ILP can be exploited
Software Pipelining Idea

iterat. 1

original loop

new loop
Software Pipelining
Prologue, Body, and Epilogue

PROLOGUE
load a[1]
add a[1], #1
load a[2]

EPILOGUE
store c[i]
add a[i+1], #1
load a[i+2]

original loop

new loop
Why “SW Pipelining”? 

**HW pipelining**
- Fetch (i+2)
- Decode (i+1)
- Execute (i)

**SW pipelining**
- Load (i+2)
- Add (i+1)
- Store (i)

**Instructions**
advancing in parallel

**Iterations**
advancing in parallel
Pipelining Result

- One output sample produced every 4 cycles and minimal resources

```c
shift_reg[2] = shift_reg[1];
shift_reg[1] = shift_reg[0];
shift_reg[0] = x;

acc = 0;
for (i = 10; i >= 0; i--) {
    #pragma HLS pipeline
    acc += shift_reg[i] * c[i];
}
y = acc;
```
Loop Restructuring as with VLIWs

- Initiation Interval = 5
  - element $i$
  - element $i - 4$

- Initiation Interval = 1
  - element $i$
  - element $i - 4$
Classic HLS and VLIW Compilation

• Striking resemblance of the two undertakings
  – Both try to produce a static schedule of operations
  – Both try to reduce to a minimum control decisions

• Both suffer from similar limitations: they cope poorly with variability including variable latency operations, uncertain events—such as memory dependencies, unpredictable control flow (see later)

• Both impose burdens onto the user: decisions on how and where to apply optimizations are not self-evident, depend on the particular combination of user constraints (note that the solution space is much wider for HLS), and thus are often left to users through code restructuring or pragmas
Extent of Programming Language Support

• Complete support for C/C++? Not quite:
  – No dynamic memory allocation (no malloc(), etc.)
    • Research work on providing such primitives for FPGA accelerators in high-end systems, for instance
  – No recursion
  – Limited use of pointers-to-pointers
  – No system calls (no printf(), etc.)
  – Other limitations related to the ability to determine critical details (e.g., function interfaces) at compile time

• Details vary from HLS tool to HLS tool
  – Perhaps similarly to the early days of logic synthesis (which part of VHDL is supported and with what exact meaning?)
1. Moore, spatial computing, FPGAs, HLS
2. Why hardware?
   Why do we expect hardware to be better than software...
3. HLS #1: Instruction set extensions
   Not really what most people see as HLS
4. HLS #2: Classic statically scheduled HLS
   The most common form of HLS today
5. HLS #3: Dataflow circuits
   The out-of-order superscalar processor equivalent in HLS
6. Some caveats in form of conclusions
High-level Synthesis and Static Scheduling

- **High-level synthesis (HLS)** may be the future of reconfigurable computing
  - Design circuits from high-level programming languages

- As seen before, classic HLS relies on **static schedules**
  - Each operation executes at a cycle fixed at synthesis time

- Scheduling dictated by compile-time information
  - Maximum parallelism in regular applications
  - Limited parallelism when information unavailable at compile time (i.e., latency, memory or control dependencies)
The Limitations of Static Scheduling

for (i=0; i<N; i++) {
    hist[x[i]] = hist[x[i]] + weight[i];
}

• Static scheduling (standard HLS tool)
  – Inferior when memory accesses cannot be disambiguated at compile time

  ![Dynamic Scheduling Diagram](image1)

• Dynamic scheduling
  – Maximum parallelism: Only serialize memory accesses on actual dependencies

  ![Dynamic Scheduling Diagram](image2)
Statically vs. Dynamically Scheduled

- **Statically Scheduled**
  - "Compiler does the job"
- **Dynamically Scheduled**
  - "Hardware does the job"

**Computer Architecture**

- **VLIW Processors**
  - Great for some embedded applications
    - (expert developers, heavy manual code refactoring and optimizations, etc.)

- **Out-of-Order Superscalar Processors**
  - Catastrophic for general purpose
    - (out-of-the-box compilation fails to deliver high performance)
## Statically vs. Dynamically Scheduled

<table>
<thead>
<tr>
<th>Computer Architecture</th>
<th>High-Level Synthesis</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Statically Scheduled</strong></td>
<td><strong>Dynamically Scheduled</strong></td>
</tr>
<tr>
<td>VLIW Processors</td>
<td>Out-of-Order Superscalar Processors</td>
</tr>
<tr>
<td>Traditional HLS</td>
<td>???</td>
</tr>
</tbody>
</table>

- Statically Scheduled $\rightarrow$ “Compiler does the job”
- Dynamically Scheduled $\rightarrow$ “Hardware does the job”
A Different Way to Do HLS

- Refrain from triggering the operations through a centralized pre-planned controller
- **Make scheduling decisions at runtime**: as soon as all conditions for execution are satisfied, an operation starts
A Different Way to Do HLS

• **Asynchronous circuits**: operators triggered when inputs are available
  – Budiu et al. Dataflow: A complement to superscalar. ISPASS’05.

• Dataflow, latency-insensitive, elastic: the synchronous version of it
  – Cortadella et al. Synthesis of synchronous elastic architectures. DAC’06.
  – Carloni et al. Theory of latency-insensitive design. TCAD’01.
  – Jacobson et al. Synchronous interlocked pipelines. ASYNC’02.

How to create dataflow circuits from high-level programs?
Dataflow Circuits

- Example using the **SELF (Synchronous ELastic Flow)** protocol
  - Cortadella et al. Synthesis of synchronous elastic architectures. DAC’06.
- Every component communicates via a pair of handshake signals
- The data is propagated from component to component as soon as memory and control dependencies are resolved
Dataflow Components

Functional units

Load ports

ST

Memory subsystem

Memory interface

BUFF

FIFO

Buffer

FIFO

ST

LD

LD

address

address

address

address
Dataflow Components

- Fork
- Join
- Branch
- Merge
Dataflow Components

- Fork
- Join
- Branch
- Merge
Dataflow Components
Dataflow Components

- Fork
- Join
- Branch
- Merge
Dataflow Components

• Although inspired by asynchronous circuits, elastic circuits are strictly synchronous and perfectly adapted to traditional VLSI and FPGA flows.
Synthesizing Dataflow Circuits

- C to intermediate graph representation
  - LLVM compiler framework

```c
for (i=0; i<N; i++) {
    hist[x[i]] = hist[x[i]] + weight[i];
}
```
Synthesizing Dataflow Circuits

- Constructing the datapath

```c
for (i=0; i<N; i++) {
    hist[x[i]] = hist[x[i]] + weight[i];
}
```

Each operator corresponds to a functional unit
Synthesizing Dataflow Circuits

- Implementing control flow

for (i=0; i<N; i++) {
    hist[x[i]] = hist[x[i]] + weight[i];
}

A Merge for each variable entering the BB
Synthesizing Dataflow Circuits

• Implementing control flow

for (i=0; i<N; i++) {
    hist[x[i]] = hist[x[i]] + weight[i];
}

A Branch for each variable exiting the BB
Synthesizing Dataflow Circuits

• Inserting Forks

for (i=0; i<N; i++) {
    hist[x[i]] = hist[x[i]] + weight[i];
}

A Fork after every node with multiple successors
for (i=0; i<N; i++) {
    hist[x[i]] = hist[x[i]] + weight[i];
}

But where are the registers?!
Adding Buffers

• Buffers and circuit functionality

Buffer insertion does not affect circuit functionality
Adding Buffers

- Buffers and circuit functionality
  - Buffer insertion does not affect circuit functionality

- Buffers and avoiding deadlock
  - Each combinational loop in the circuit needs to contain at least one buffer
Adding Buffers

Start: \( i = 0 \)

1. Fork
2. LD \( x[i] \)
3. Fork
4. LD weight\( [i] \)
5. Fork
6. LD hist\( [x[i]] \)
7. Fork
8. St hist\( [x[i]] \)
9. Fork
10. Exit: \( i = N \)

Combining 4 stages:

- LD \( x[i] \)
- LD weight\( [i] \)
- LD hist\( [x[i]] \)
- St hist\( [x[i]] \)

Resulting in:

Br

Combining (comb.)

\( i = 0 \)
Adding Buffers

Two combinational loops

Start: \( i=0 \)

Fork

\( \text{LD } x[i] \)

Fork

\( \text{LD weight}[i] \)

Fork

\( \text{LD hist}[x[i]] \)

4 stages

\( \text{ST hist}[x[i]] \)

1 comb.

\( + \)

\( \text{LD hist}[x[i]] \)

\( N \)
Adding Buffers

Start: i=0

Fork

Mg

LD x[i]

LD hist[x[i]]

ST hist[x[i]]

4 stages

<

1

Fork

LD weight[i]

Fork

Exit: i=N

4 stages comb.

Adding Buffers
Adding Buffers

Backpressure from slow paths prevents pipelining
Optimizing Performance

Start: $i=0$

- **Mg**
  - **Buff**
    - **Fork**
      - **LD $x[i]$**
      - **LD weight[i]**
      - **LD hist$[x[i]]$**
      - **ST hist$[x[i]]$**
    - **Exit: $i=N$**
  - **Stages**
    - **Comb.**
    - **N**
      - **Br**

$+ 1$
Optimizing Performance

Insert FIFOs into slow paths

Start: i=0

LD x[i]

Mg

Buff

Fork

Fork

LD hist[x[i]]

FIFO

LD weight[i]

1

N

<

comb.

4 stages

ST hist[x[i]]

FIFO

LD hist[x[i]]

Exit: i=N

Br
Optimizing Performance

BEFORE (without FIFOs)

Start: i=0

Mg

Buff

Fork

LD x[i]

Fork

FIFO

LD weight[i]

Fork

LD hist[x[i]]

FIFO

LD hist[x[i]]

ST hist[x[i]]

Fork

ST hist[x[i]]

Exit: i=N

4 stages

comb.

Fork

LD hist[x[i]]

4 stages

comb.

N

<

N

<

Exit: i=N
Optimizing Performance

**NOW (with FIFOs)**

- **Start:** $i = 0$
- **Mg**
- **Fork**
- **LD $x[i]$**
- **FIFO**
- **LD $\text{hist}[i]$**
- **Fork**
- **LD $\text{weight}[i]$**
- **Fork**
- **ST $\text{hist}[x[i]]$**
- **4 stages**
- **Fork**
- **Exit:** $i = N$

**BEFORE (without FIFOs)**

- **Start:** $i = 0$
- **Mg**
- **Fork**
- **LD $x[i]$**
- **Fork**
- **LD $\text{hist}[i]$**
- **Fork**
- **LD $\text{weight}[i]$**
- **Fork**
- **LD $\text{hist}[x[i]]$**
- **4 stages**
- **Fork**
- **Exit:** $i = N$
Optimizing Performance
Optimizing Performance

RAW dependency not honored!

What about memory?
The Problem with Memory

- Accesses on a single LD or ST node happen in order but, across all LD and ST nodes, can happen in arbitrary order.
- A very similar problem happens in out-of-order superscalar processors.

Approx. rules to issue memory accesses:

- The oldest memory access in the queue is always safe to execute.
- Stores always wait to be the oldest memory access to execute.
- A load can execute ahead of preceding accesses if (1) the address for all older stores is known and (2) no store conflicts with it (i.e., no store writes to the same address).
- [memory bypass, etc.]
We Need a Load-Store Queue (LSQ)!

• Traditional processor LSQs allocate memory instructions **in program order**

```
loop: lw $t1, 0($t0)
lw $t2, 0($t1)
mul $t2, $t2, $t3
sw $t2, 0($t0)
addi $t1, $t1, 4
bne $t5, $t1, loop
```
We Need a Load-Store Queue (LSQ)!

- Traditional processor LSQs allocate memory instructions in program order

```assembly
loop: lw $t1, 0($t0)
lw $t2, 0($t1)
mul $t2, $t2, $t3
sw $t2, 0($t0)
addi $t1, $t1, 4
bne $t5, $t1, loop
```

- Dataflow circuits have no notion of program order

How to supply program order to the LSQ?
Basic Idea

- An LSQ for dataflow circuits whose only difference is in the allocation policy:
  - Static knowledge of memory accesses program order inside each basic block
  - Dynamic knowledge of the sequence of basic blocks from the dataflow circuit
Dataflow Circuit with the LSQ

High-throughput pipeline with memory dependencies honored
Experimental Results

• Resource utilization and execution time of the dataflow designs, normalized to the corresponding static designs produced by Vivado HLS.
Nonspeculative vs. Speculative System

float d=0.0; x=100.0; int i=0;

do {
    d = a[i] + b[i];
    i++;
}
while (d<x);

1: a[0]=50.0; b[0]=30.0
2: a[1]=40.0; b[1]=40.0
Nonspeculative vs. Speculative System

float d=0.0; x=100.0; int i=0;
do {
    d = a[i] + b[i];
    i++;
}while (d<x);

1: a[0]=50.0; b[0]=30.0
2: a[1]=40.0; b[1]=40.0
Speculative Tokens

• Extend dataflow components with a speculative tag
  – An additional bit propagated with the data or OR’ed from all inputs
Speculation in Dataflow Circuits

• Contain speculation in a region of the circuit delimited by special components
  – Issue speculative tokens (pieces of data which might or might not be correct)
  – Squash and replay in case of misspeculation
Results

• Timing and resources: traditional HLS (Static) and dataflow circuits with speculation (Speculative)
  – Cases where dynamic scheduling on its own cannot achieve high parallelism

What to Expect from Dynamic HLS?

• Two hopes derived from the VLIW vs. OoO analogy:
  – Significantly better performance in control dominated applications with poorly predictable memory accesses
  – Better out-of-the-box performance

• The former is almost certain, the second less so

• A major issue is the hardware overhead of supporting dynamic schedules
  – Probably tolerable for the bulk of the circuits
  – Yet, LSQs represent quite tangible overheads, esp. in FPGAs (but could be hardened there)

• Probably statically-scheduled HLS remains the best choice for classic DSP-like applications
Outline

1. Moore, spatial computing, FPGAs, HLS
2. Why hardware?
   Why do we expect hardware to be better than software...
3. HLS #1: Instruction set extensions
   Not really what most people see as HLS
4. HLS #2: Classic statically scheduled HLS
   The most common form of HLS today
5. HLS #3: Dataflow circuits
   The out-of-order superscalar processor equivalent in HLS
6. Some caveats in form of conclusions
Takeaway

• High-Level Synthesis is a **fairly mature field** with a number of options available

• Of commercial interest **virtually only for FPGAs**; ASIC designers seem to stick almost exclusively to RTL because they want to get everything they can from technology

• Still, it is **not really meant to** (nor in fact does) **give access to FPGAs for software programmers**; most of the available options speed-up development for hardware engineers who more or less know what they want
Caveat #1

The accelerator design must compensate for the FPGA reconfigurability

- Processors exploit amazingly well transistors in a given technology
- GPUs do that too and also extract a huge lot of parallelism—when the application fits the architecture
- ASICs are for the lucky few, but if an application is sufficiently important (e.g., Google’s TPUs), they are unbeatable, almost by definition
- FPGAs must fight the overhead of reconfigurability (one order of magnitude slower and bigger?) before one can gain!
Caveat #2

“It’s the memory, stupid!”
Dick Sites, 1996

- HLS is pretty good at designing the computational part of accelerators
- Most of the performance depends on moving data efficiently
- Even the simplest aspects of this are hard or hopeless for HLS compilers (memory disambiguation, etc.)
- Few tools to help designing application-specific memory systems
- Expect to plan data movement by hand—and to code it in RTL...
**Caveat #3**

HLS may give you great kernels but does not give you full accelerators

- HLS tools extract well fine grain parallelism (akin to ILP)
- They have no real way to go significantly beyond that
- Successful languages to express more parallelism are tied to very particular computational patterns (e.g., CUDA, OpenCL)
- HLS tools have embraced some of these—but are GPUs not better when you can use these languages proficiently?
- Manual design in RTL (with ad-hoc use of HLS, perhaps) seems the only way of achieving truly competitive accelerators today
References

HLS #1

• P. Ienne and R. Leupers, eds., *Customizable Embedded Processors*, MK, 2006

HLS #2


HLS #3

• L. Josipović, R. Ghosal, and P. Ienne, *Dynamically Scheduled High-level Synthesis*, International Symposium on Field-Programmable Gate Arrays, February 2018
• L. Josipović, A. Guerrieri, and P. Ienne, *Speculative Dataflow Circuits*, International Symposium on Field-Programmable Gate Arrays, February 2019