

# Synthesis of Software Kernels in Hardware

Vitalik Nikolyenko  
The University of Sydney

November 19, 2010

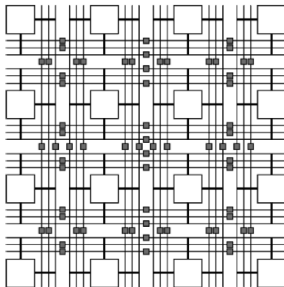
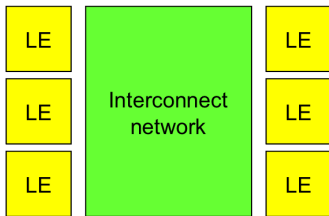
# Software development

Software development approaches:

- ▶ hardware-to-software
  - ▶ developing for a particular *fixed* architecture
  - ▶ dominating approach in software construction
- ▶ software-to-hardware
  - ▶ designing an architecture for a *specific* task
  - ▶ targets application requirements
  - ▶ achieves substantial benefits over the traditional approach
  - ▶ possible with reconfigurable computing

# Reconfigurable computing

- ▶ refers to some form of hardware programmability
- ▶ many technologies that allow for hardware programmability
  - ▶ e.g., PLAs, MPGAs, FPGAs, etc
  - ▶ FPGAs are the most common underlying technology
- ▶ FPGAs are programmable logic devices:



# Reconfigurable computing (cont.)

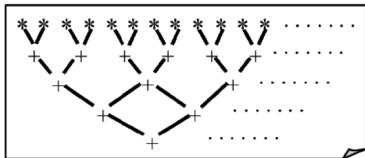
- ▶ reconfigurable systems are often coupled with a processor
  - ▶ processor controls the reconfigurable logic
  - ▶ executes portions of applications that cannot be accelerated efficiently
  - ▶ performs hardware/software partitioning
  - ▶ scheduling
- ▶ fills the gap between hardware and software
  - ▶ software flexibility
  - ▶ hardware performance
    - ▶ fixed data path
    - ▶ no instructions (controlled by registers)

# Example

```
for (i=0; i < 128; i++)  
  accum += c[i] * x[i]  
..  
..
```



*1000 s of  
instructions*



*$\log 128 = 7$  cycles  
Speedup > 100X*

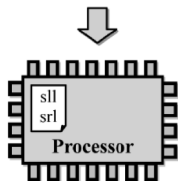
[Lysecky et al., 2006]

# Example

Bit-level operations:

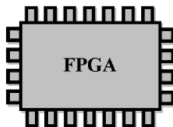
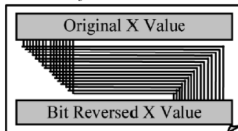
*C Code for Bit Reversal*

```
x = (x >>16) | (x <<16);  
x = ((x >> 8) & 0x00ff00ff) | ((x << 8) & 0xff00ff00);  
x = ((x >> 4) & 0x0f0f0f0f) | ((x << 4) & 0xf0f0f0f0);  
x = ((x >> 2) & 0x33333333) | ((x << 2) & 0xcccccccc);  
x = ((x >> 1) & 0x55555555) | ((x << 1) & 0xaaaaaaaa);
```



*64 instructions,  
32 to 128 cycles*

*Hardware for Bit Reversal*



*1 cycle, Speedup  
of 32X to 128X*

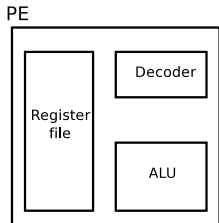
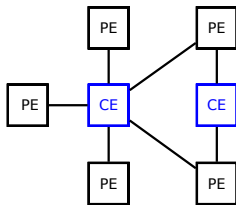
[Lysecky et al., 2006]

# Target architecture

Underlying dataflow architecture:

- ▶ limited control flow
  - ▶ e.g., loops
- ▶ shared interconnect
  - ▶ island-style, network-on-chip routing
- ▶ local register-file

Register transfer instructions are required!



# Hardware-software partitioning

## Syntax

- ▶ limited C/C++ syntax
- ▶ explicit kernel definition
- ▶ special data-types

```
1: void exec_path(register_bank &rb)
2: {
3:     // set the size of the
4:     // register bank
5:     rb.set_size(1);
6:
7:     // fill the register bank
8:     // with some values
9:     ddg_float a(1.6);
10:    rb.store(0,a);
11: }
12:
13: void kernel(register_bank &rb)
14: {
15:     ddg_float a = rb.load_float(0);
16:     ddg_float b = ddg_float( 5.6 );
17:     ddg_int    c = a + b;
18:     ddg_float d = c * b;
19:     ddg_float e = a + b;
20:     ddg_float f = a / e;
21:
22:     for(int i=0;i<5;i++) {
23:         a = a * f;
24:     }
25:     rb.store(0,a);
26: }
```

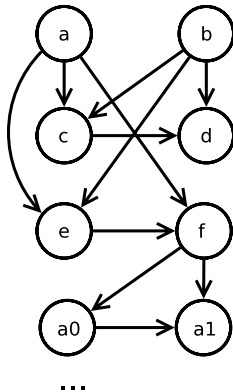


# Hardware-software partitioning

## Data-dependency graph

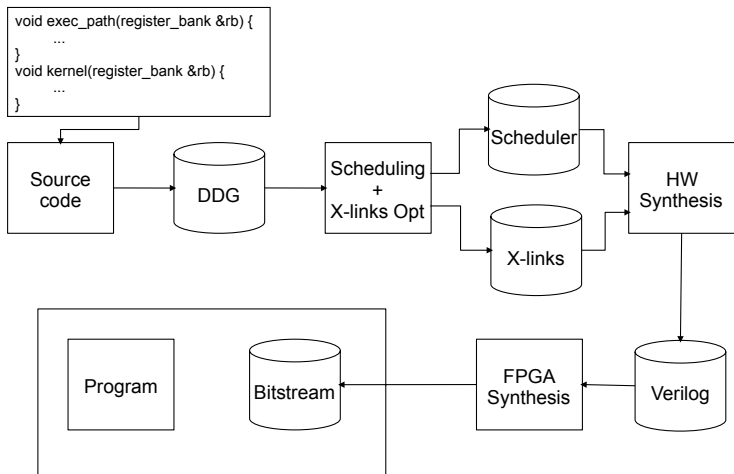
```
void kernel(register_bank &rb)
{
    ddg_float a = rb.load_float(0);
    ddg_float b = ddg_float( 5.6 );
    ddg_int    c = a + b;
    ddg_float d = c * b;
    ddg_float e = a + b;
    ddg_float f = a / e;

    for(int i=0;i<5;i++) {
        a = a * f;
    }
    rb.store(0,a);
}
```



# Hardware synthesis flow

Synthesising kernels in hardware



# Scheduling

## Scheduling:

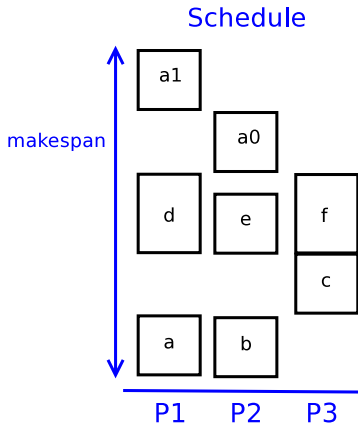
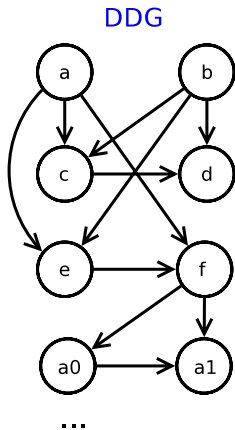
- ▶ assigns operations to processors
- ▶ ensures that data dependencies between operations are enforced
- ▶ minimises the total execution time (makespan)
- ▶ NP-hard for multiprocessor systems  
[Garey and Johnson, 1990]

## List scheduling approach:

- ▶ heuristic to order tasks in a list and then greedily distribute them among processors
- ▶ 2-approximation

# Scheduling

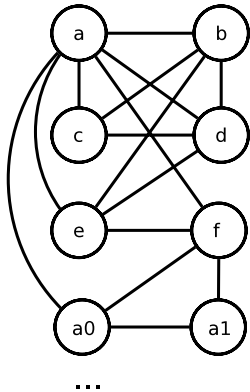
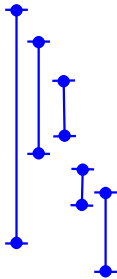
## Example



# Register allocation

## Example

```
a = 1.6  
b = 5.6  
c = a + b  
d = c * b  
e = a + b  
f = a / e  
a0 = a * f  
a1 = a0 * f
```



# Register allocation

Optimal register allocation in linear time:

- ▶ graph coloring
  - ▶ introducing communication instructions into the DDG
  - ▶ constructing and partitioning the interval graph
  - ▶ using Lex-BFS search to find a perfect ordering
  - ▶ applying greedy coloring

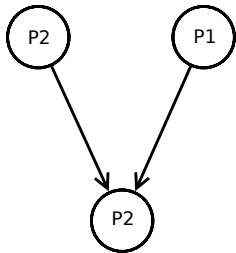


Figure: data dependencies

# Register allocation

Optimal register allocation in linear time:

- ▶ graph coloring
  - ▶ introducing communication instructions into the DDG
  - ▶ constructing and partitioning the interval graph
  - ▶ using Lex-BFS search to find a perfect ordering
  - ▶ applying greedy coloring

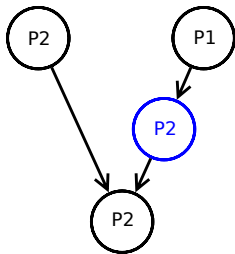


Figure: data dependencies

# Cross-links

Register transfer instructions (communication instructions):

- ▶ require opening of new cross-links (communication links between PEs)
- ▶ *expensive*
  - ▶ logic required (transistors)
  - ▶ makespan
- ▶ add extra complexity to the scheduling problem



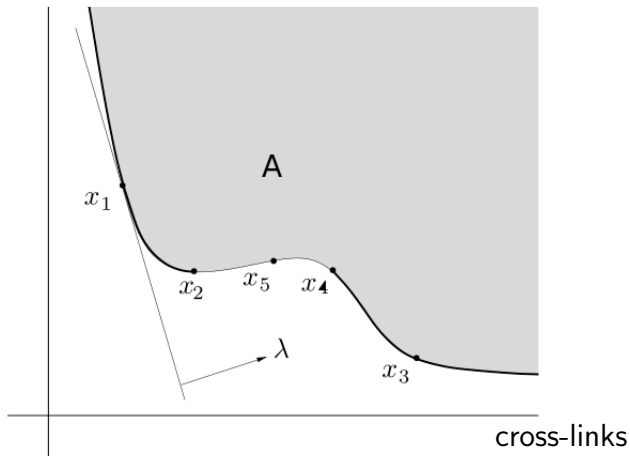
# Optimisation question

A multi-objective function of producing an optimal schedule:

- ▶ minimising the makespan
- ▶ reducing the number of cross-links
- ▶ can be described using the concept of Pareto optimality [Boyd and Vandenberghe, 2004]

# Optimisation trade-offs

makespan






# Plan

- ▶ construct a mathematical formulation considering the constraints
- ▶ use Integer Linear Programming (ILP) to find an optimal solution
  - ▶ not practical for large problem sizes
  - ▶ a good yardstick to compare how good available heuristics perform

# Summary

New prototype system presented for the automatic hardware/software partitioning

- ▶ easy to use
- ▶ provides significant performance (\*hopefully)
- ▶ 2-approximation scheduling algorithm
- ▶ optimal register allocation in linear time

-  Boyd, S. and Vandenberghe, L. (2004).  
*Convex Optimization.*
-  Garey, M. R. and Johnson, D. S. (1990).  
*Computers and Intractability; A Guide to the Theory of NP-Completeness.*
-  Lysecky, R., Stitt, G., and Vahid, F. (2006).  
Warp processors.