Binary Synthesis with Runtime Dependence Validation

Shaoyi Cheng, Qijing Huang, John Wawrzynek
University of California, Berkeley

FPGA Clouds

Hardware as a Service

Performance

Ease of Use

Cost

Alibaba Cloud

Microsoft Azure

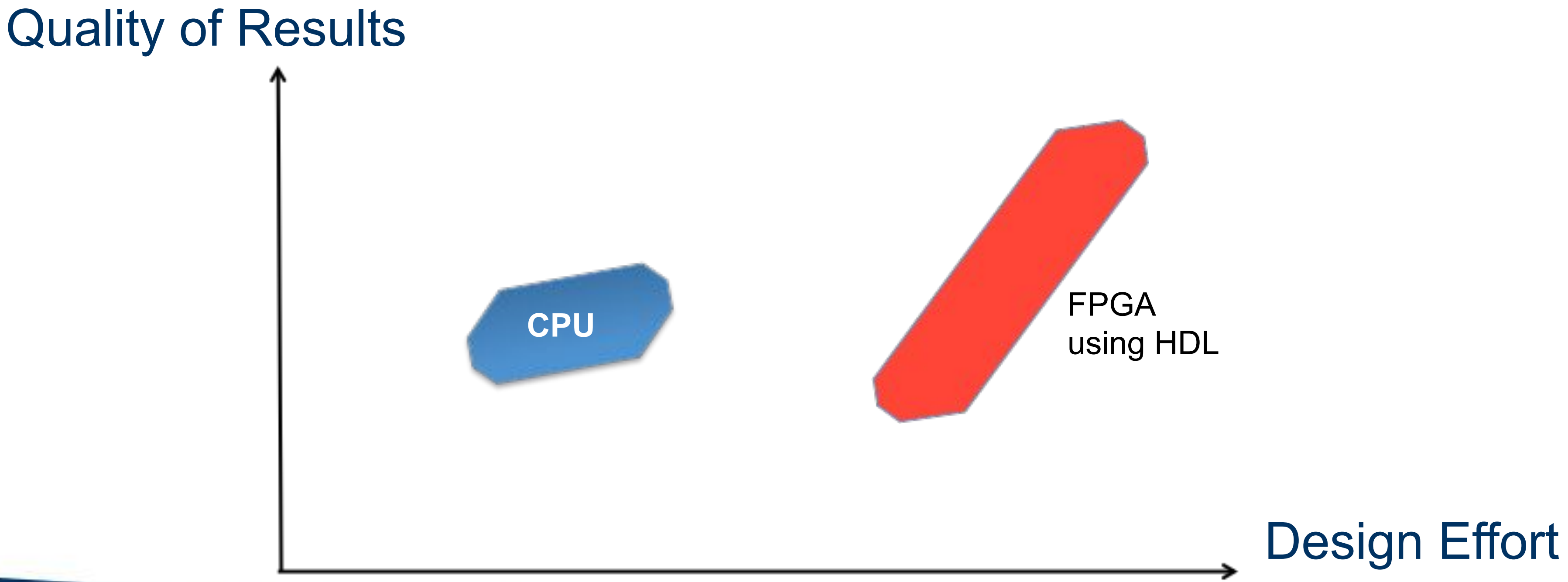
Baidu Cloud

Tencent Cloud

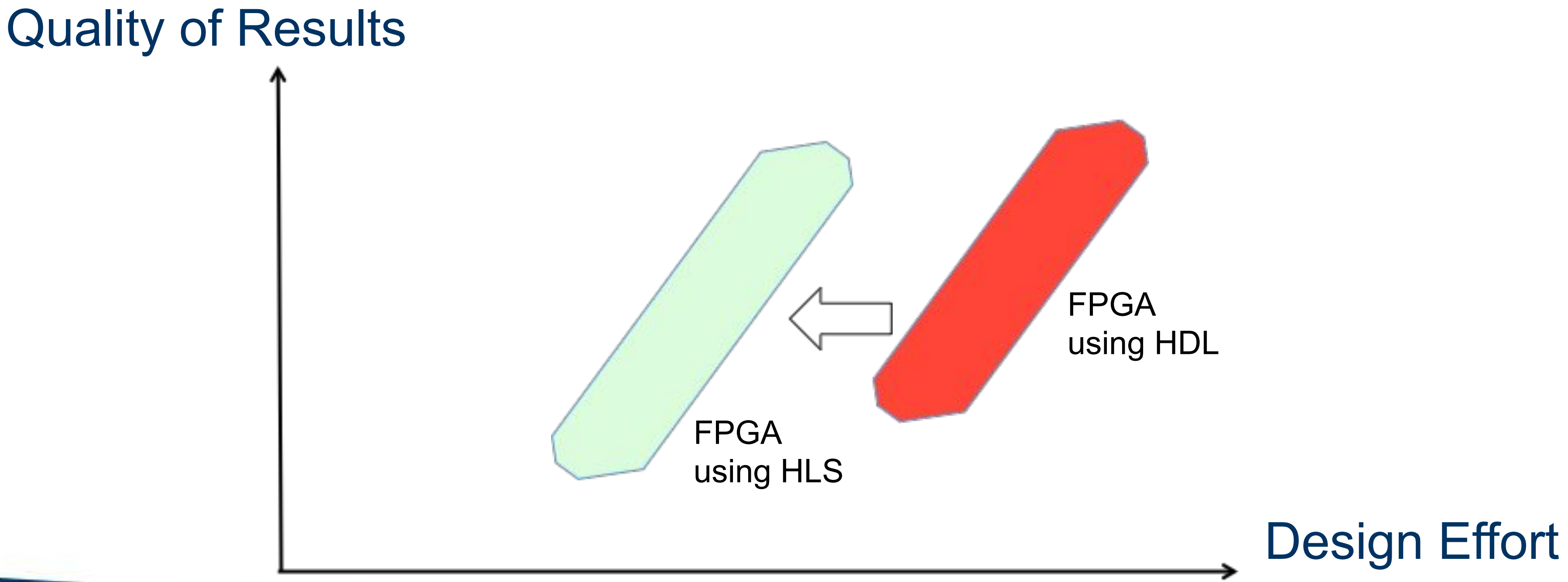
aws

HUAWEI

A Hard-to-program Device

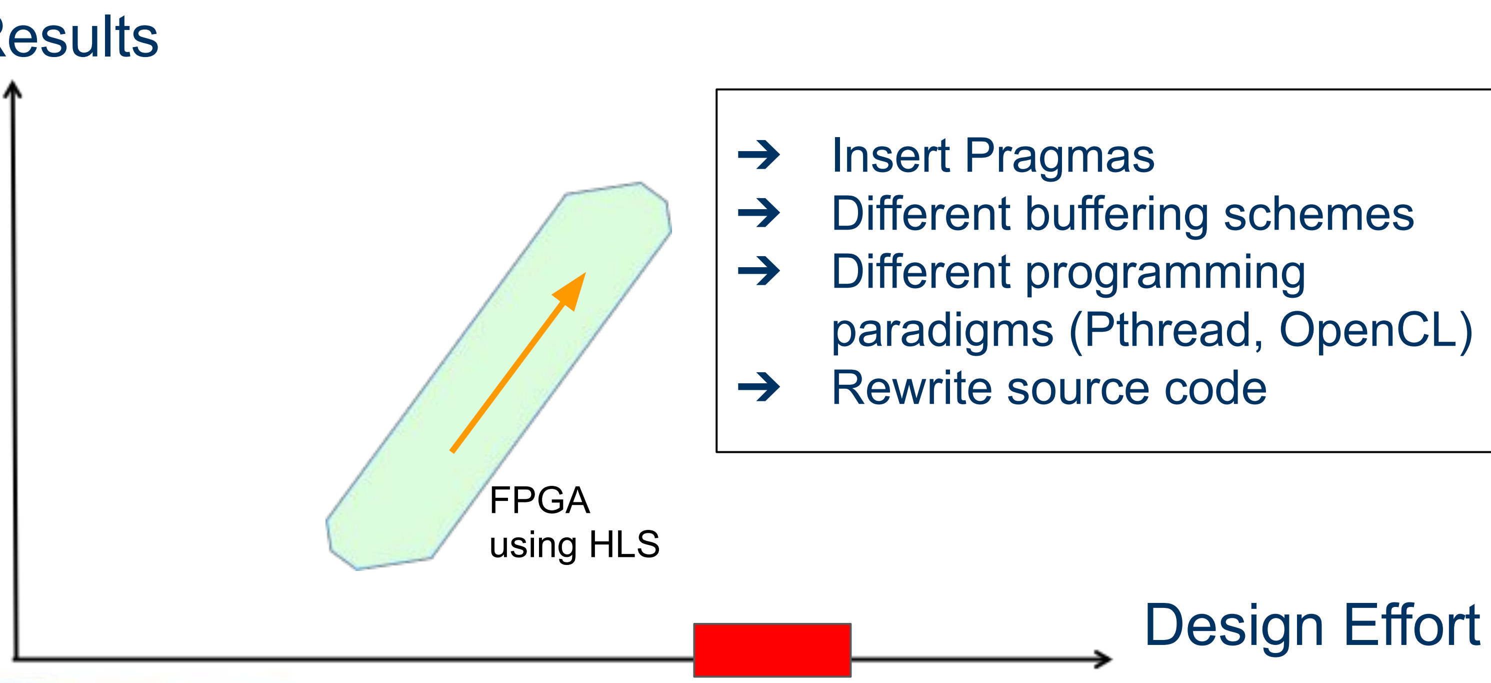


A Hard-to-program Device



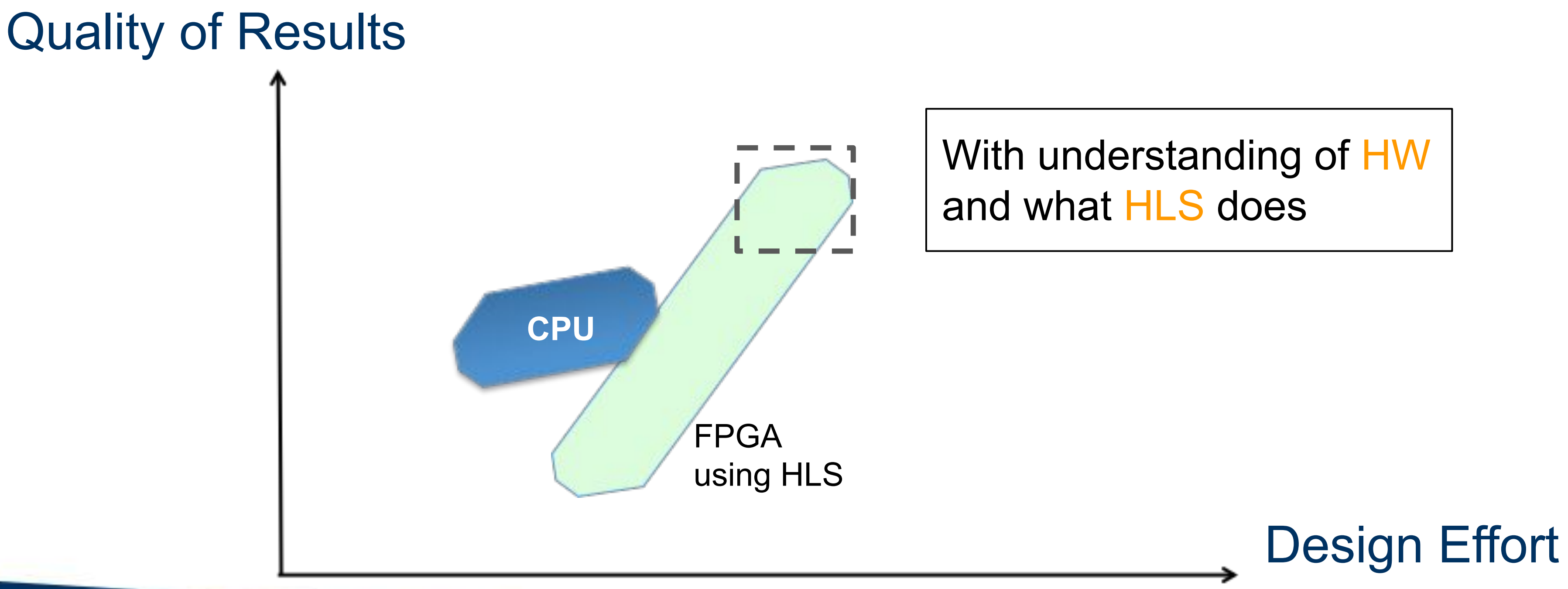
Berkeley
UNIVERSITY OF CALIFORNIA

A Hard-to-program Device



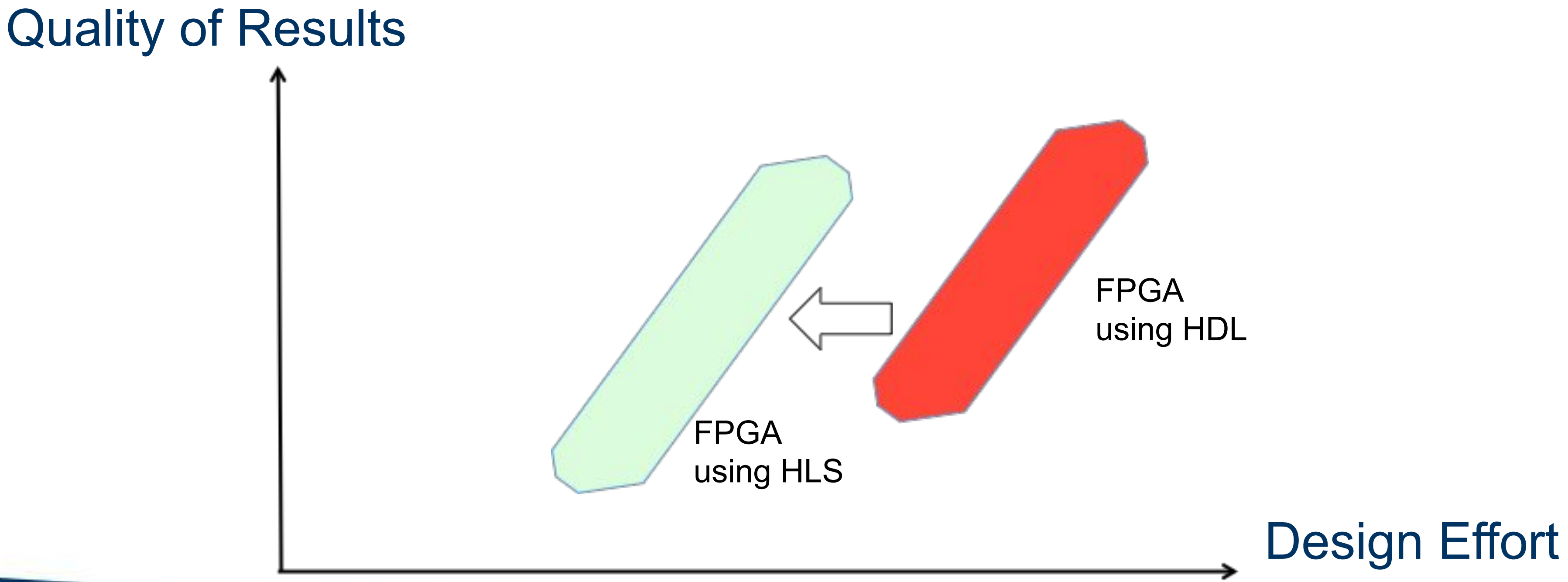
- ➔ Insert Pragmas
- ➔ Different buffering schemes
- ➔ Different programming paradigms (Pthread, OpenCL)
- ➔ Rewrite source code

A Hard-to-program Device



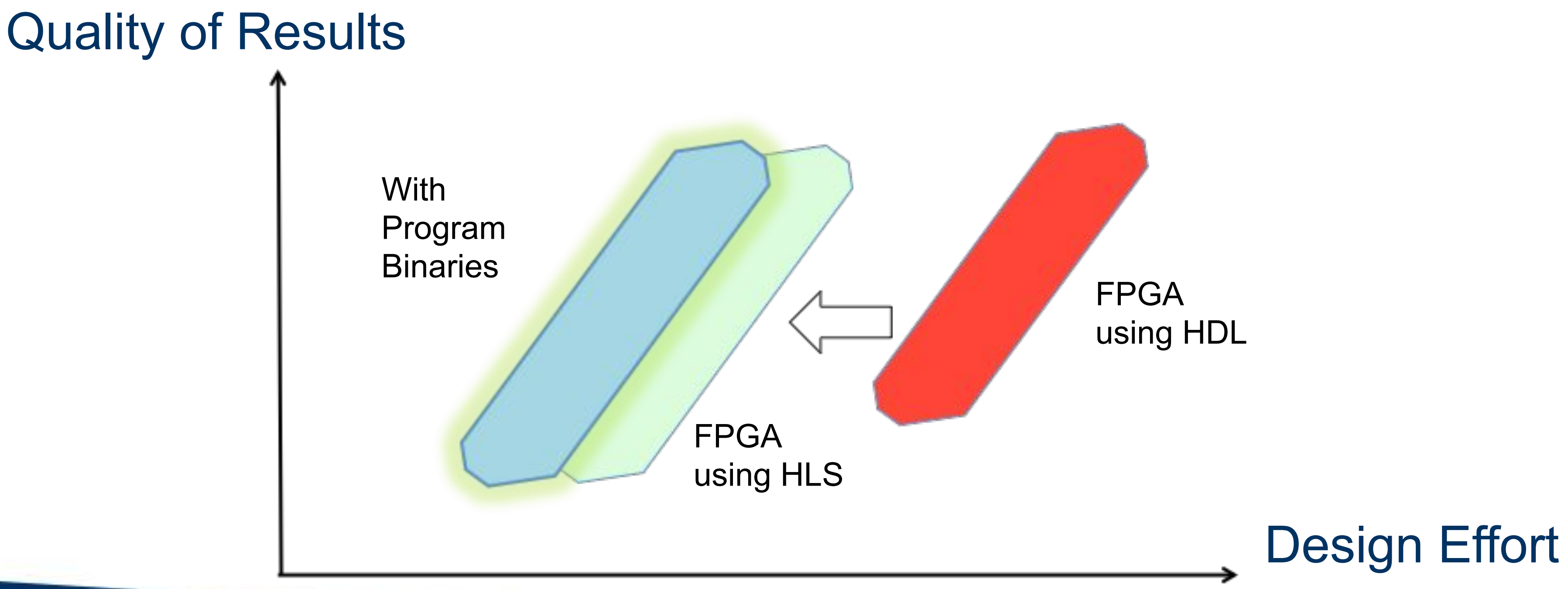
With understanding of HW and what HLS does

A Hard-to-program Device

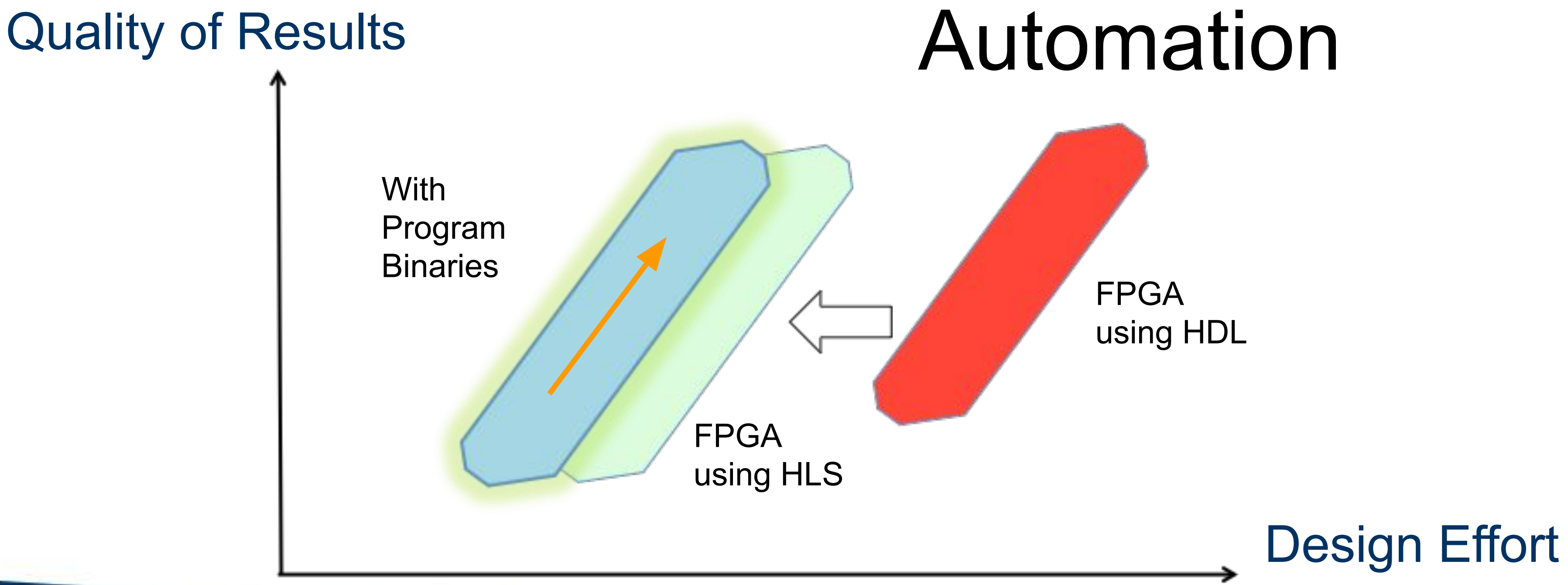


Berkeley UNIVERSITY OF CALIFORNIA

A Hard-to-program Device



A Hard-to-program Device

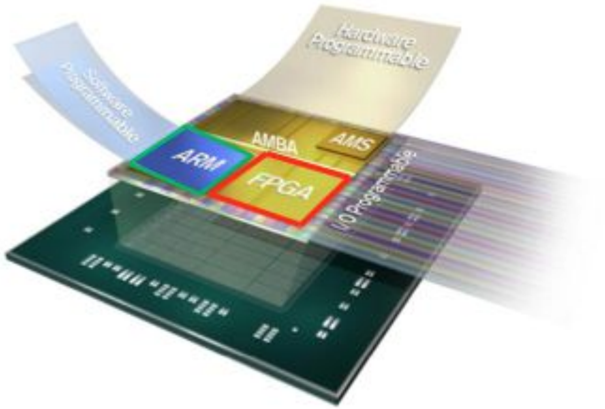


User-transparent Accelerator Integration

Program Binaries

- Non-invasive
- Libraries without source code
- Languages not currently supported by HLS

Push the limit in ease of use:
Program binaries as design entry



Berkeley
UNIVERSITY OF CALIFORNIA

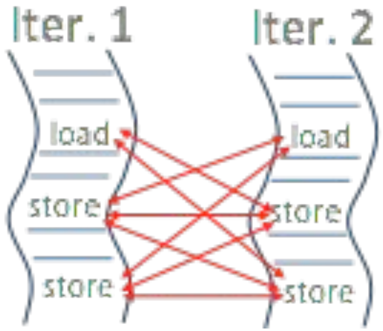
Binary Synthesis

- Input: Program binaries and execution profiles
  - Assume no other user input
  - Leverage techniques from parallelizing compilers
  - Automatically exploit coarse-grained parallelism

- Targeting platforms: With shared CPU and FPGA address space
  - Existing FPGA SoCs -> ZynQ Platform
  - FPGA+Xeon Platform

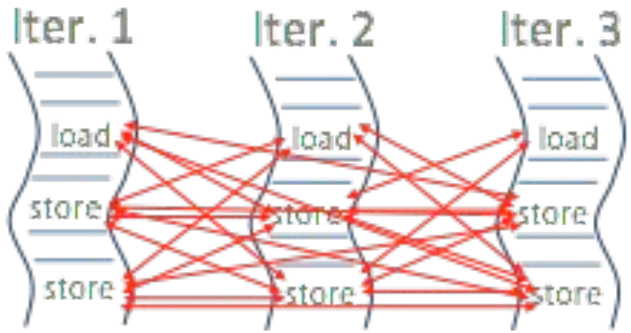
Coarse-grained Parallelism

When the memory locations accessed in loop iterations do not intersect



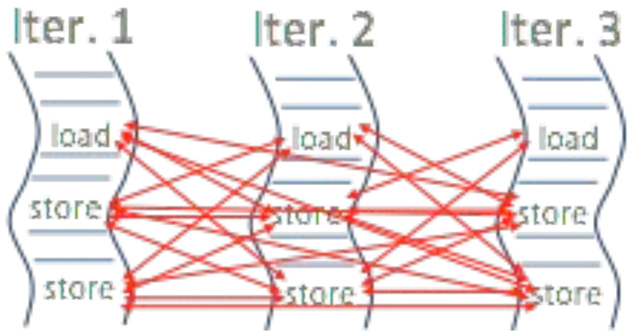
Coarse-grained Parallelism

When the memory locations accessed in loop iterations do not intersect:



Coarse-grained Parallelism

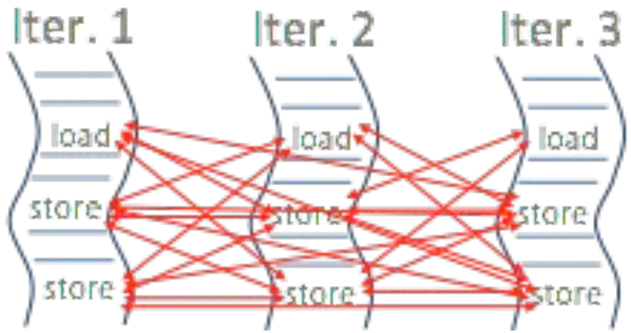
When the memory locations accessed in loop iterations do not intersect:



Potentially many address comparisons for aggressive parallelization.

Coarse-grained Parallelism

When the memory locations accessed in loop iterations do not intersect



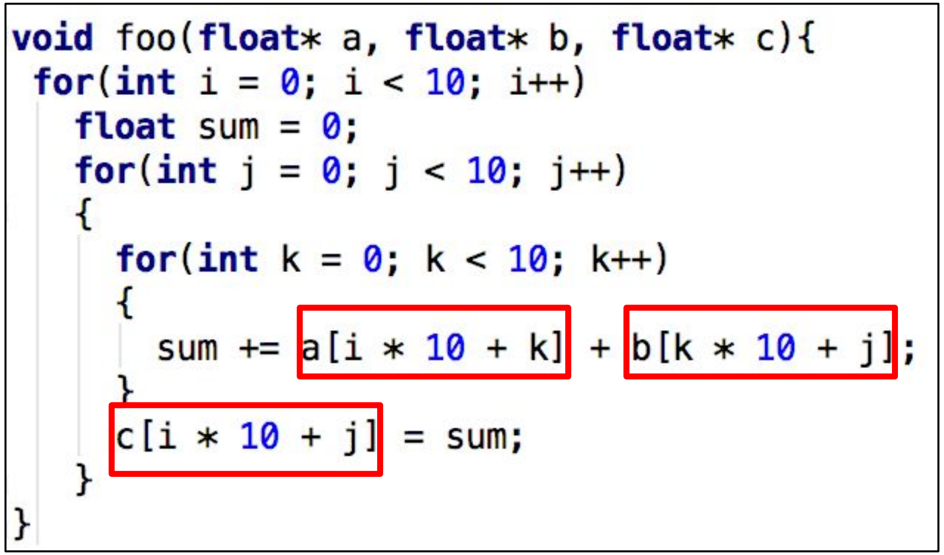
Our Target:

Regular computation kernels:

➔ Affine array references

➔ Whether they intersect can be determined statically

Affine Array References



```
void foo(float* a, float* b, float* c){
 for(int i = 0; i < 10; i++)
   float sum = 0;
   for(int j = 0; j < 10; j++)
   {
     for(int k = 0; k < 10; k++)
     {
       sum += a[i * 10 + k] + b[k * 10 + j];
     }
     c[i * 10 + j] = sum;
   }
}
```

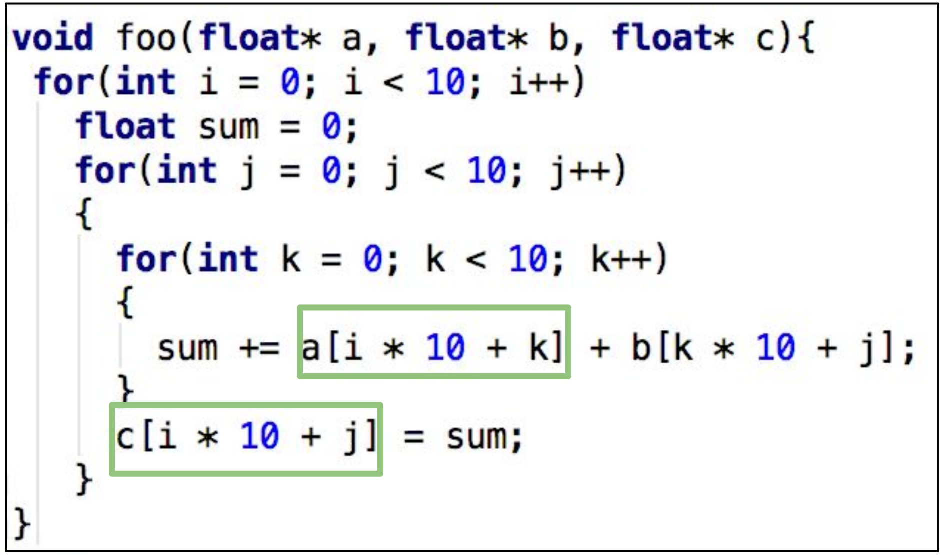
Affine function of the indices

Diophantine Equation:

$$a_0+a_1x_1+a_2x_2+\ldots a_nx_n - (b_0+b_1y_1+b_2y_2+\ldots b_ny_n) = 0 \text{ ?}$$

Affine Array References

Omits the "**restrict**" keyword



```
void foo(float* a, float* b, float* c){
 for(int i = 0; i < 10; i++)
   float sum = 0;
   for(int j = 0; j < 10; j++)
   {
     for(int k = 0; k < 10; k++)
     {
       sum += a[i * 10 + k] + b[k * 10 + j];
     }
     c[i * 10 + j] = sum;
   }
}
```

Diophantine Equation:

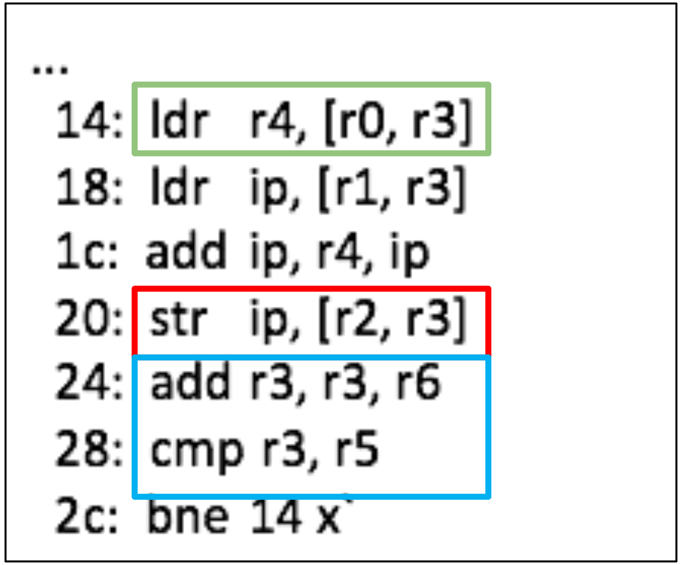
$$a_0+a_1x_1+a_2x_2+\dots a_nx_n - (b_0+b_1y_1+b_2y_2+\dots b_ny_n) = 0 \ ?$$

c + 4*i*10 + 4*j = a + 4*i'*10 + 4*k' ?

Existing techniques for identifying parallelism:

➔ GCD test, **Banerjee's test**, Omega test etc.

Challenges



```
...
  14: ldr   r4, [r0, r3]
  18: ldr   ip, [r1, r3]
  1c: add  ip, r4, ip
  20: str   ip, [r2, r3]
  24: add r3, r3, r6
  28: cmp r3, r5
  2c: bne 14 x
```

Iteration Space:

$[r3_{initial}, r3+r6 \ldots, r5]$

Equation:

Not Statically Solvable!

Offline Profiling

```
...
 14:  ldr   r4, [r0, r3]
 18:  ldr   ip, [r1, r3]
 1c:  add  ip, r4, ip
 20:  str   ip, [r2, r3]
 24:  add r3, r3, r6
 28:  cmp r3, r5
 2c:  bne 14
```

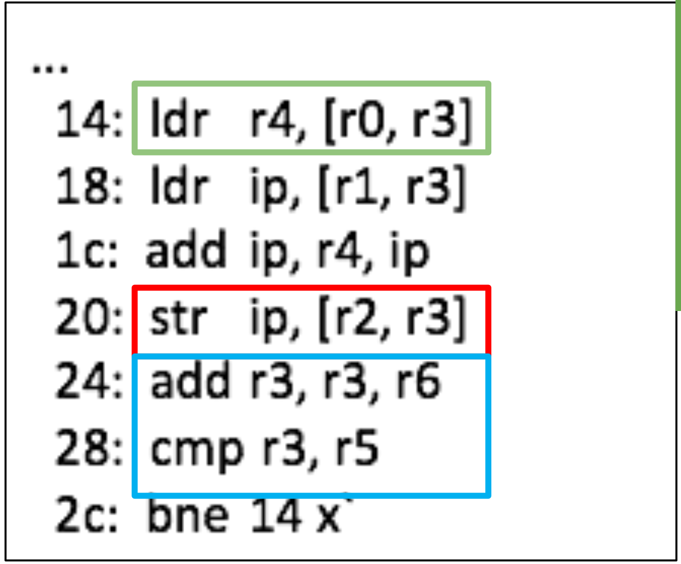
From Past Profiles

```
#define r0_INIT = 0x10000
#define r2_INIT = 0x20000
#define r3_INIT = 0
....
    r0 = r0_INIT;
    r1 = r1_INIT;
    r6 = r6_INIT;
...
BB14:
    r4 = *(r0+r3);
    ip = *(r1+r3);
    ip = ip+r4;
    *(r2+r3) = ip;
    r3 = r3+r6;
    If(r3!=r5)
       goto BB14;
```

Dependency Analysis
(Banerjee's Test)

Parallelizable?

Runtime Validation

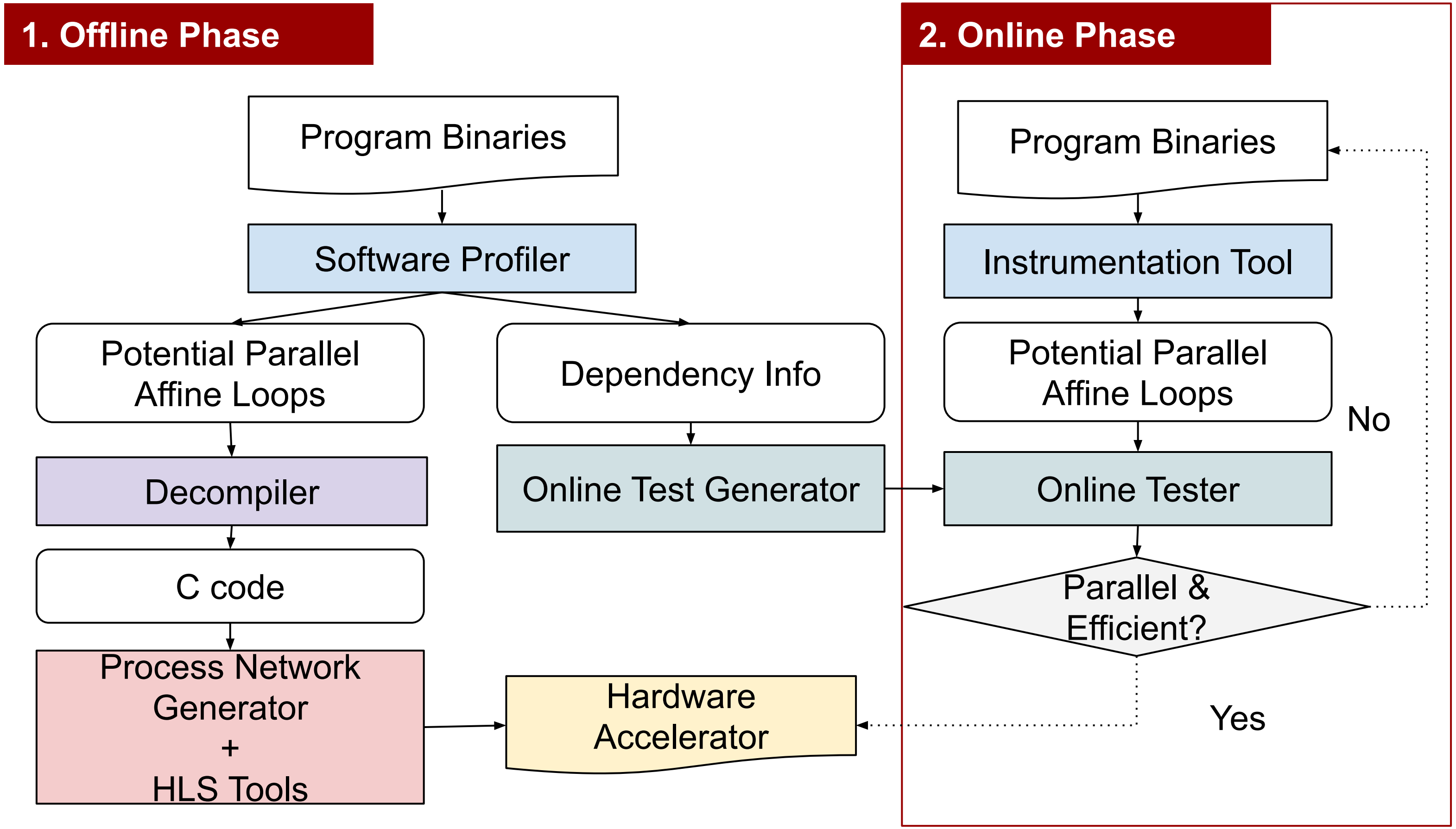


```
...
 14: ldr   r4, [r0, r3]
 18: ldr   ip, [r1, r3]
 1c: add  ip, r4, ip
 20: str   ip, [r2, r3]
 24: add r3, r3, r6
 28: cmp r3, r5
 2c: bne 14 x
```

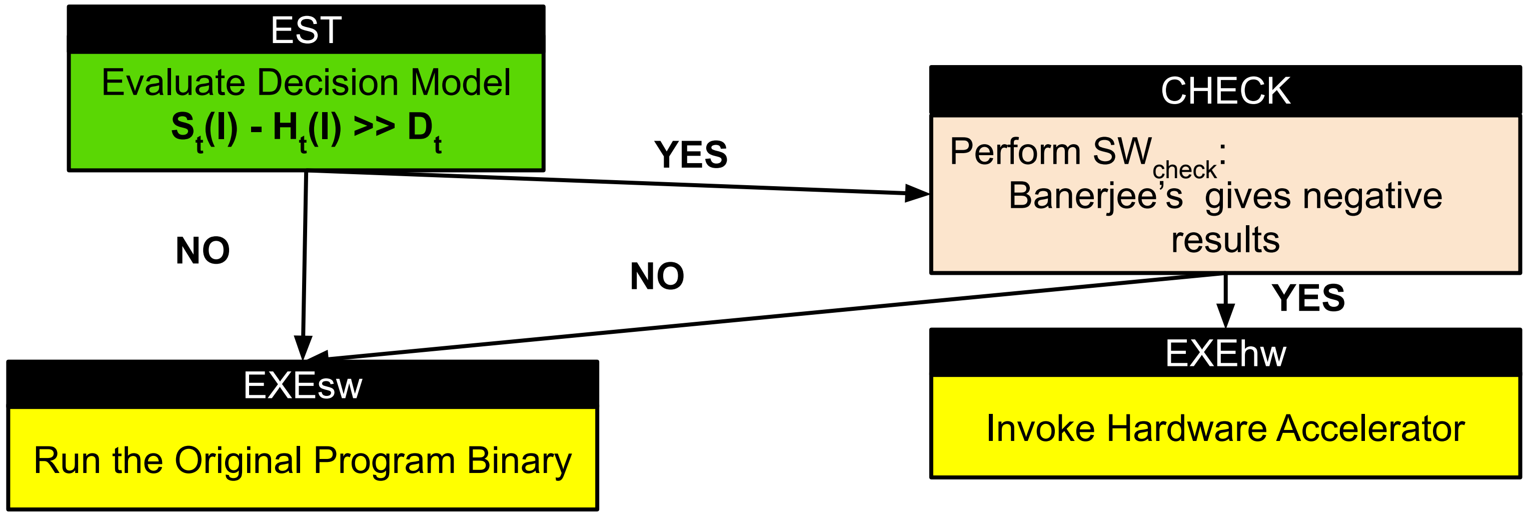
Rerun the Banerjee's test during the actual execution, before test accelerator starts

What if r0+r6 = r2 when the FPGA accelerator is invoked?

A Two-phase Approach



Online Decision Model

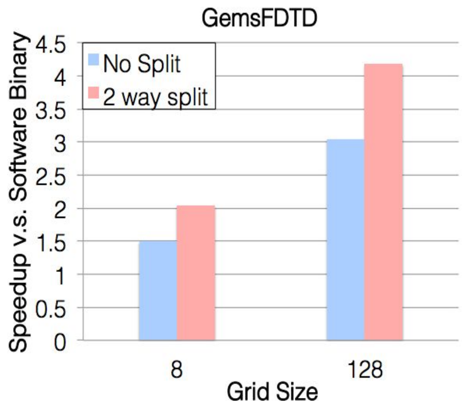


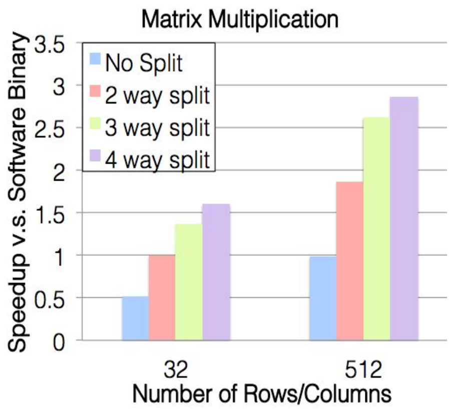
	Steps Taken	**Performance Benefits**
Best	**EST->CHECK->EXEhw**	$S_t(I) - H_t(I) - D_t - E_t$
Early Abort	**EST->EXEsw**	$- E_t$
Worst Case	**EST->CHECK->EXEsw**	$- D_t - E_t$

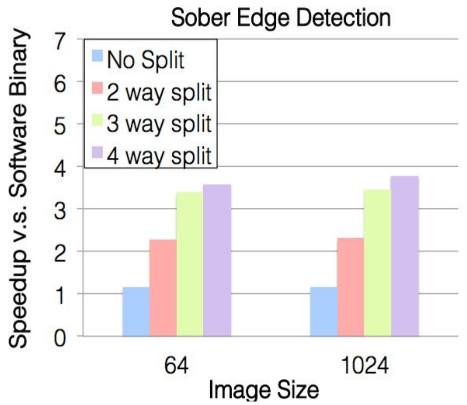
Experiments

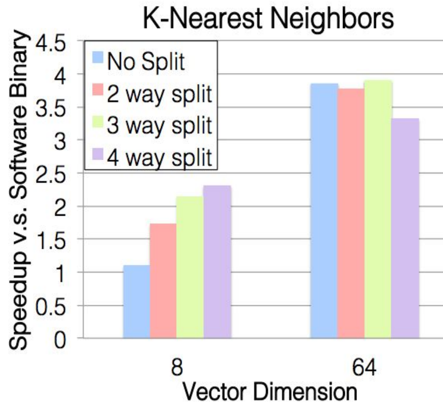
- ➔ Validate the binary based flow
- ➔ Four regular kernels:
  - ◆ Gems FDTD Simulation, Matrix Multiplication, Sober Edge Detector, K-Nearest Neighbors
- ➔ Vary the degree of parallelism to fill up the area
- ➔ Vary the number of ports
- ➔ Quantify the runtime overhead
- ➔ Compare against Software performance on Arm Cortex-A9 running at 667MHz

Results - Performance



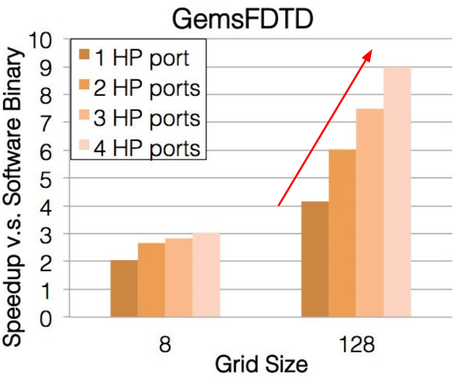


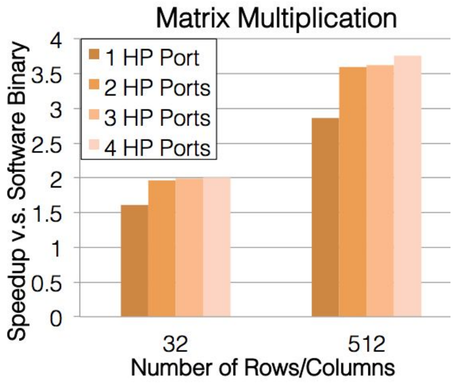


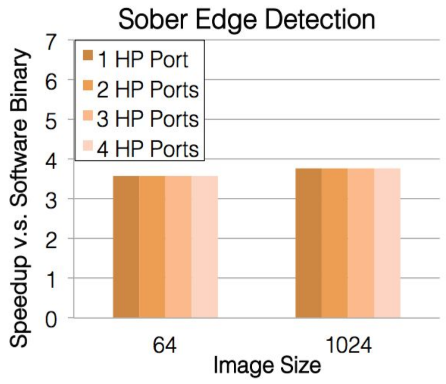


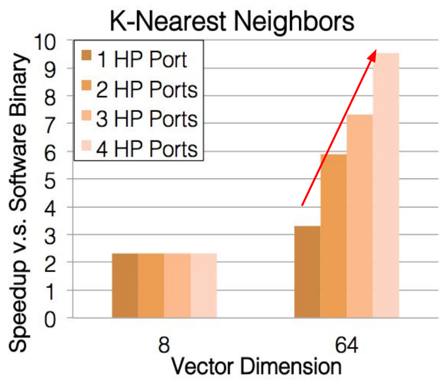
- More aggressive parallelization -> higher performance
- Convolution is compute-bound

Results - Performance





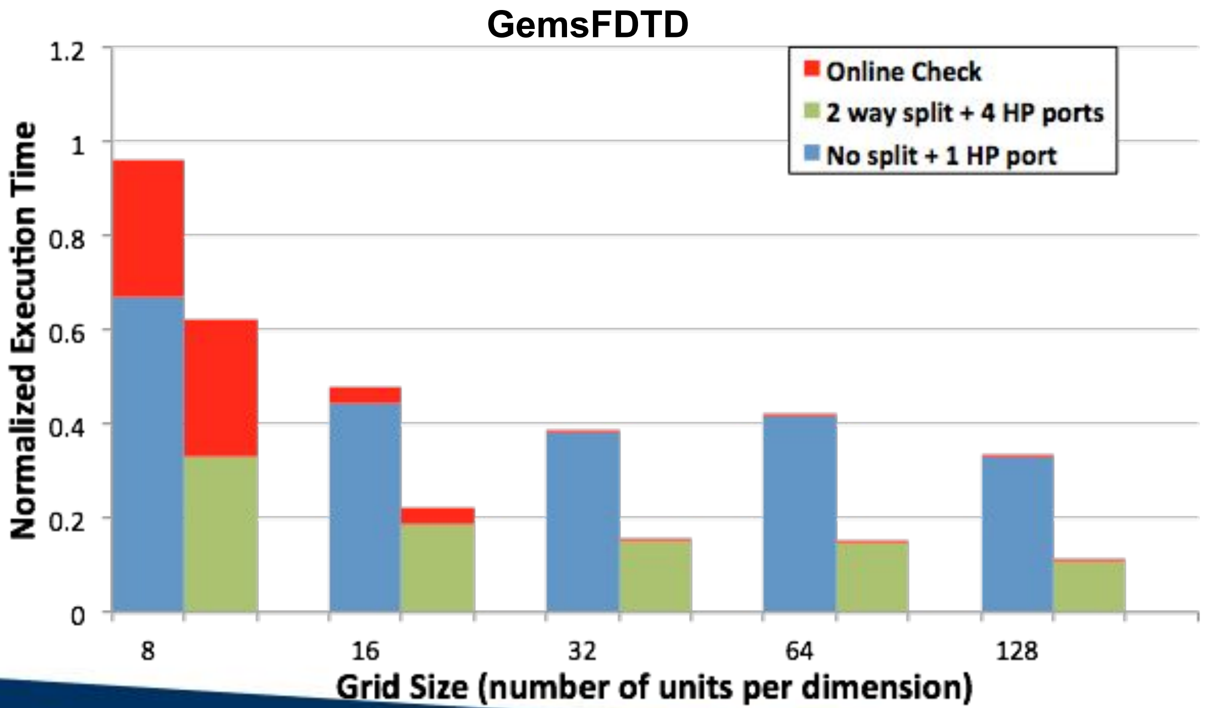




- Open more ports to memory -> higher performance
- GemsFDTD and KNN in larger size is memory-bound

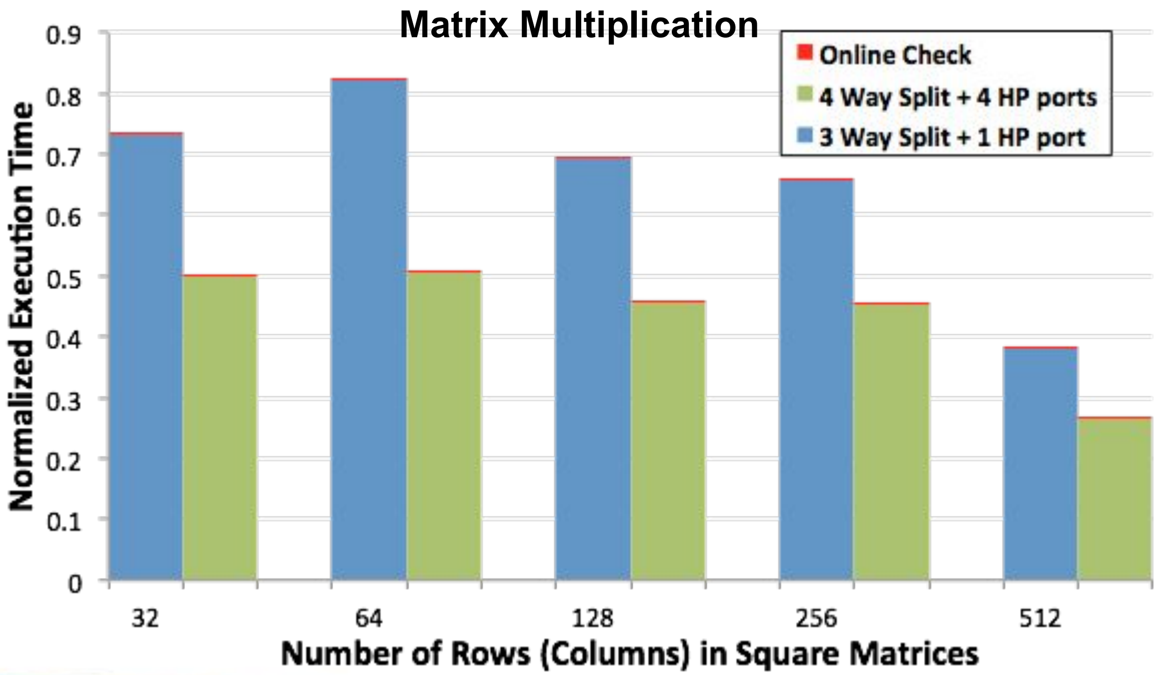
Up to 9.5x Speedup

Results - Runtime Overhead



Insignificant overhead except for small grid size

Results - Runtime Overhead



Negligible Overhead

Summary

Our binary synthesis flow:

1. Complements existing HLS flow
2. Generates design with good performance
3. Offloads computation in user transparent way
4. Improves the ease of use for FPGA

Future work:

- More aggressive optimizations with runtime validation methodology
- More complicated runtime generated binaries
- Apply to other heterogeneous platforms

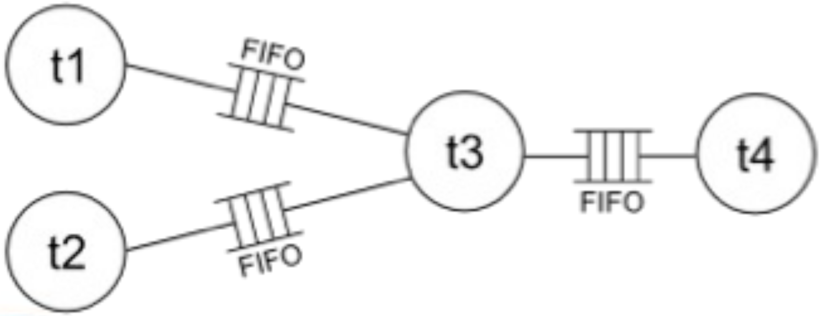
Thanks!

Questions

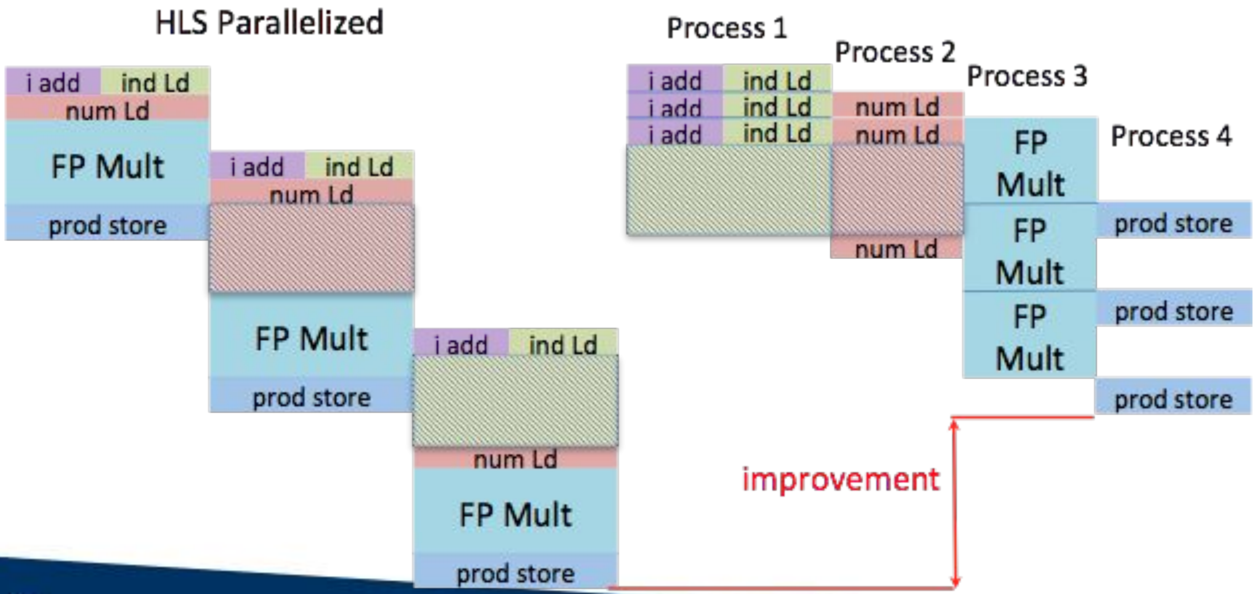
Backup Slides

Process Network Generation

- ➔ Transform sequential programs to process networks
  - ◆ A parallel model of computation
  - ◆ Processes executed concurrently
  - ◆ Processes connected by FIFOs
    - Blocking reads
    - Blocking writes in real implementation



from Centralized to Distributed Control



System Integration

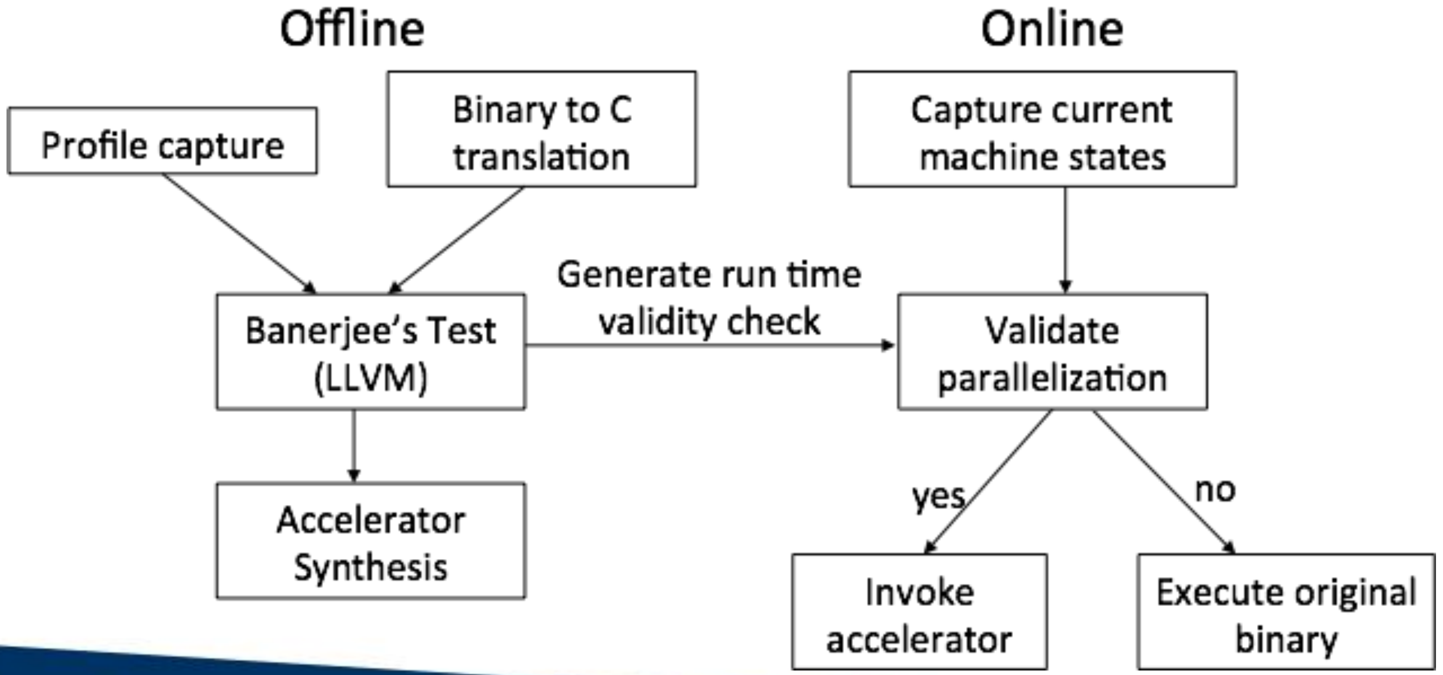
- ➔ Leverage existing API for binary instrumentation (Dyinst):
  1. Supports both static and dynamic modification of binaries
  2. Abstract away the details of the low-level machine code

- ➔ Package the validation routine and accelerator invocation into a function
- ➔ Redirect the binary to call the new function

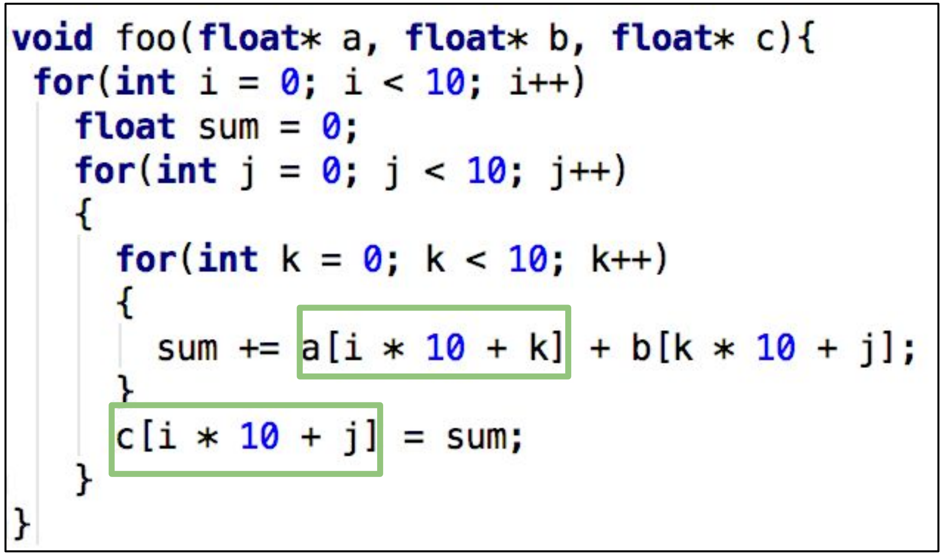
May apply to other type of heterogeneous computing platforms.

Berkeley
UNIVERSITY OF CALIFORNIA

A Two-phase Approach



Affine Array References



```
void foo(float* a, float* b, float* c){
 for(int i = 0; i < 10; i++)
   float sum = 0;
   for(int j = 0; j < 10; j++)
   {
     for(int k = 0; k < 10; k++)
     {
       sum += a[i * 10 + k] + b[k * 10 + j];
     }
     c[i * 10 + j] = sum;
   }
}
```

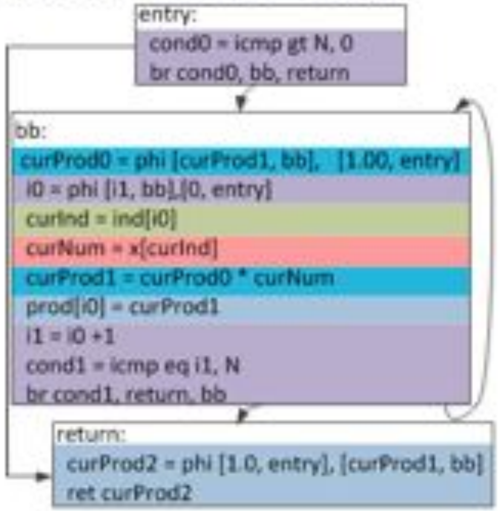
Diophantine Equation:

$$a_0+a_1x_1+a_2x_2+\dots a_nx_n - (b_0+b_1y_1+b_2y_2+\dots b_ny_n) = 0 \text{ ?}$$

c + 4*i*10 + 4*j = a + 4*i’*10 + 4*k’

Partitioning Algorithm for FPGA

```
float foo (float* x, float* product, int* ind)
{
  float prod = 1.0;
  for(int i=0; i<N; i++)
  {
     int curInd = ind[i];
     float curNum = x[ind];
     prod = prod * curNum;
     product[i] = prod;
  }
  return curProd;
}
```



```
entry:
  cond0 = icmp gt N, 0
  br cond0, bb, return

bb:
  curProd0 = phi [curProd1, bb],  [1.00, entry]
  i0 = phi [i1, bb],[0, entry]
  curInd = ind[i0]
  curNum = x[curInd]
  curProd1 = curProd0 * curNum
  prod[i0] = curProd1
  i1 = i0 +1
  cond1 = icmp eq i1, N
  br cond1, return, bb

return:
  curProd2 = phi [1.0, entry], [curProd1, bb]
  ret curProd2
```

- Collapse strongly connected components in CDFG
  - Obtain a directed acyclic graph (DAG)
- Perform topological sort on DAG
- Add a subgraph “boundary” when appropriate
  - After long latency SCCs. E.g. SCCs with multiply/floating point arith.
  - After long memory operation.