ESE532: System-on-a-Chip Architecture

Day 10: October 4, 2017
Coding HLS for Accelerators

Previously

- We can describe computational operations in C
  - Primitive operations (add, sub, multiply, and, or)
  - Dataflow graphs primitives
  - To bit level
  - Conditionals and loops
  - Memory reads/writes
  - Function abstraction
- Need to avoid
  - Recursive function calls, dynamic allocation

Perspectives

- Here's a computation we want to describe
  - How can we use C to describe
  - What do we need to watch to avoid getting tangled up in sequential C semantics
- Here's an arbitrary piece of C code
  - What will the compiler be able to do with it?
- What would it take to write a C-to-gates compiler
- What are pitfalls inherent in the C language?

Today

- Pipelining loops
- Pragmas in Vivado HLS C
- Avoiding bottlenecks feeding data in Vivado HLS C
- Streaming hardware operations

Message

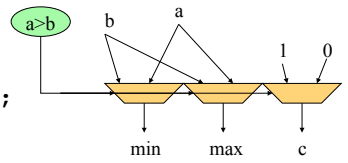
- Can specify HW computation in C
- Vivado HLS gives control over how design mapped (area-time, streaming…)
- Code may need some care and stylization to feed data efficiently
- Read Design Productivity Guide (UG 1197)
  - C-based IP development
- Reference Vivado HLS Users Guide (902)
  - Design Optimization

Finish up Mux Conversion

(about what compiler will do;
Not much about what developer does)

Mux conversion for simple conditionals

```
max=a;
min=a;
if (a>b)
  {min=b;
   c=1;}
else
  {max=b;
   c=0;}
```



- May (re)define many values on each branch.

Mux Conversion and Memory

- What might go wrong if we mux-converted the following:
- If (cond)
  - *a=0
- Else
  - *b=0

Mux Conversion and Memory

- What might go wrong if we mux-converted the following:
- If (cond)
  - *a=0
- Else
  - *b=0
- Don't want memory operations in non-taken branch to occur.

Mux Conversion and Memory

- If (cond)
  - *a=0
- Else
  - *b=0
- Don't want memory operations in non-taken branch to occur.
- **Conclude:** cannot mux-convert blocks with memory operations (without additional care)

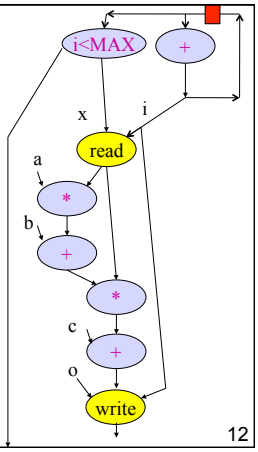
Optimizations can expect compiler to do

- Constant propagation: a=10; b=c[a];
- Copy propagation: a=b; c=a+d; → c=b+d;
- Constant folding: c[10*10+4]; → c[104];
- Identity Simplification: c=1*a+0; → c=a;
- Strength Reduction: c=b*2; → c=b<<1;
- Dead code elimination
- Common Subexpression Elimination:
  - C[x*100+y]=A[x*100+y]+B[x*100+y]
  - t=x*100+y; C[t]=A[t]+B[t];
- Operator sizing: for (i=0; i<100; i++) b[i]=(a&0xff+i);

Pipelining

for (i=0;i<MAX;i++)
  o[i]=(a*x[i]+b)*x[i]+c;

- If know memory operations independent
- What II?

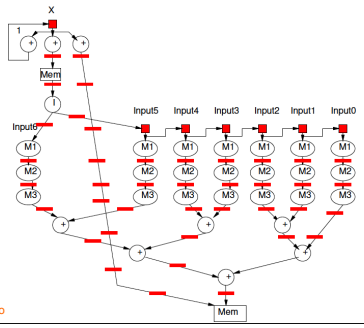


Loop Interpretations

- What does a loop describe?
  - Sequential behavior [when execute]
  - Spatial construction [when create HW]
  - Data Parallelism [sameness of compute]
- We will want to use for all 3
- Sometimes need to help the compiler understand which we want

C Loops

- Adequate to define hardware pipelines



Vivado HLS Mapping Control

Preclass 1

- What dataflow graph does this describe?

```
while(true) {
        i=read_input();
        fA(i,t1);
        fB(t1,t2);
        fC(t2,out);
        write_output(out);
}
```


Vivado HLS Pragma DATAFLOW

- Enables streaming data between functions and loops
- Allows concurrent streaming execution
- Requires data be produced/consumed sequentially
- Useful to use stream data type between functions
  - hls::stream<TYPE>


```
int data_in[N],data_out[N*256];
hls::stream<int> ystream;
short val,res,copies;
int current;

#pragma HLS dataflow

for (i=0;i<N;i++) {
   pair=data[i];
   copies=(pair>>16)&0x0ff;
   val=pair&0x0ffff;
   for (j=0;j<copies;j++)
      ystream.write(val);
   }

for (int i=0;i<N*256;i++)
  {
     ystream.read(res);
     current=current+res;
     data_out[i]=current;
  }
```

Vivado HLS Pragma PIPELINE

- Direct a function or loop to be pipelined
- Ideally start one loop or function body per cycle
  - Can control II

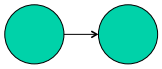

```
for (i=0;i<N;i++)
    yout=0;
    #pragma HLS PIPELINE
    for (j=0;j<K;j++)
        yout+=in[i+j]*w[j];
    y[i]=yout;
```

Which solution from preclass 2?

Dataflow and pipelining

- Dataflow allows coarse-grained pipelining among loops and functions
- Pipeline causes loop bodies to be pipelined

Dataflow and Pipelining



- Cycles with no dataflow, no pipelining?
- Dataflow only?
- Pipeline only?
- Dataflow and pipeline?

```
for (i=0;i<N;i++) {
  yout=0;
  for (j=0;j<K;j++)
    yout+=in[i+j]*w[j];
  ystream.write(yout);
  }

for (i=0;i<N;i++) {
  ystream.read(d);
  y1=max(d,y3);
  tmp=min(d,y3);
  y2=max(tmp,y2);
  tmp=min(tmp,y2);
  y3=max(tmp,y3);
  }
```

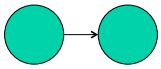

Vivado HLS Pragma UNROLL

- Unroll loop into spatial hardware
  - Can control level of unrolling
- Any loops inside a pipelined loop gets unrolled by the PIPELINE directive


```
for (i=0;i<N;i++)
    yout=0;
    #pragma HLS UNROLL
    for (j=0;j<K;j++)
        yout+=in[i+j]*w[j];
    y[i]=yout;
```

Which solution from preclass 2?

Dataflow and Pipelining



- Cycles unroll K-loop, dataflow, pipeline?

```
for (i=0;i<N;i++) {
  yout=0;
  for (j=0;j<K;j++)
    yout+=in[i+j]*w[j];
  ystream.write(yout);
  }

for (i=0;i<N;i++) {
  ystream.read(d);
  y1=max(d,y3);
  tmp=min(d,y3);
  y2=max(tmp,y2);
  tmp=min(tmp,y2);
  y3=max(tmp,y3);
  }
```


Unroll

- Can perform partial unrolling
- **#pragma HLS UNROLL factor=…**
- Use to control area-time points
  - Use of loop for spatial vs. temporal description

Vivado HLS Pragma INLINE

- Collapse function body into caller
  - Eliminates interface code
  - Allows optimization of inline code
- recursive option to inline a hierarchy
  - Maybe useful when explore granularity of accelerator

Vivado HLS Pragma ARRAY_PARTITION

- Spread out array over multiple BRAMs
  - By default placed in single BRAM
  - Use to remove memory bottleneck that prevents pipelining (limits II)

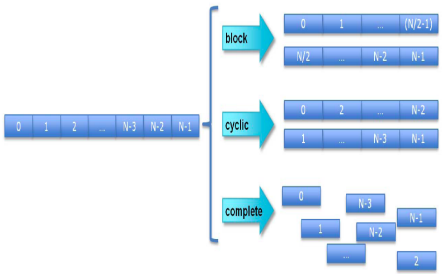
Memory Bottleneck Example

```
#include "bottleneck.h"

dout_t bottleneck(din_t mem[N]) {

  dout_t sum=0;
  int i;

  SUM_LOOP: for(i=3;i<N;i=i+4)
#pragma HLS PIPELINE
  sum += mem[i] + mem[i-1] + mem[i-2] + mem[i-3];

  return sum;
}
```

What problem if put mem in single BRAM?

 Xilinx UG1197 (2017.1) p. 50

Array Partition



 Xilinx UG902 p. 195 (145 in 2017.1 version)

Array Partition Example

#pragma ARRAY_PARTITION variable=mem cylic factor=4

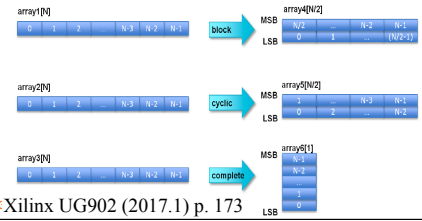
```
#include "bottleneck.h"

dout_t bottleneck(din_t mem[N]) {

  dout_t sum=0;
  int i;

  SUM_LOOP: for(i=3;i<N;i=i+4)
#pragma HLS PIPELINE
  sum += mem[i] + mem[i-1] + mem[i-2] + mem[i-3];

  return sum;
}
```


Xilinx UG902 p. 91

Vivado HLS Pragma ARRAY_RESHAPE

- Pack data into BRAM to improve access (reduce BRAMs)
  - May provide similar benefit to partitioning without using more BRAMs


```
void foo (...) {
int  array1[N];
int  array2[N];
int  array3[N];
#pragma HLS ARRAY_RESHAPE variable=array1 block factor=2 dim=1
#pragma HLS ARRAY_RESHAPE variable=array2 cycle factor=2 dim=1
#pragma HLS ARRAY_RESHAPE variable=array3 complete dim=1
...
}
```



Xilinx UG902 (2017.1) p. 173


```
#include "bottleneck.h"

dout_t bottleneck(din_t mem[N]) {

  dout_t sum=0;
  int i;

  SUM_LOOP: for(i=3;i<N;i=i+4)
#pragma HLS PIPELINE
  sum += mem[i] + mem[i-1] + mem[i-2] + mem[i-3];

  return sum;
}
```

How fix if dint_t is 16b?

Xilinx UG902 p. 91

Array Reshape Example

#pragma ARRAY_RESHAPE variable=mem cylic factor=4 dim=1
(if din_t 16b)

```
#include "bottleneck.h"

dout_t bottleneck(din_t mem[N]) {

  dout_t sum=0;
  int i;

  SUM_LOOP: for(i=3;i<N;i=i+4)
#pragma HLS PIPELINE
  sum += mem[i] + mem[i-1] + mem[i-2] + mem[i-3];

  return sum;
}
```


Xilinx UG902 p. 91

Summary

- pragmas allow us to control hardware mapping
  - How interpret loops
  - Turn area-time knobs
  - Specify how arrays get mapped to memories

Streaming Operations

Streaming Operations

- Functions can have stream inputs and outputs
  - Must pass a pointers

    hls::stream<Type> &strm
- Have expressiveness to define hardware streaming operation pipelines


```
void stream_filter (
           hls::stream<uint16_t> &strm_out,
           hls::stream<uint16_t> &strm_in
         )
  while(true) {
  yout=0;
  Input5=Input6;
  Input4=Input5;
  Input3=Input4;
  Input2=Input3;
  Input1=Input2;
  Input0=Input1;
  strm_in.read(Input0);
  Sum = Coefficients_0 * Input0 +
        Coefficients_1 * Input1 +
        Coefficients_2 * Input2 +
        Coefficients_3 * Input3 +
        Coefficients_4 * Input4 +
        Coefficients_5 * Input5 +
        Coefficients_6 * Input6;
  strm_out.write(Sum>>8);
  }
```


Big Ideas

- Can specify HW computation in C
- Create streaming operations
  - Run on processor or FPGA
- Vivado HLS gives control over how map to hardware
  - Area-time point

Admin

- Fall Break
- Back on Monday
- HW5 due 10/13