Christopher A. Wood
caw4567@rit.edu

Optimizing Impulse C Applications

Code Stages

- Impulse C compiler targets code stages
  - Individual (and independent) blocks of logic that can be executed in parallel
- Code statements with dependencies are placed in a single code block
- Memory (or other resource) access are placed in a single code block

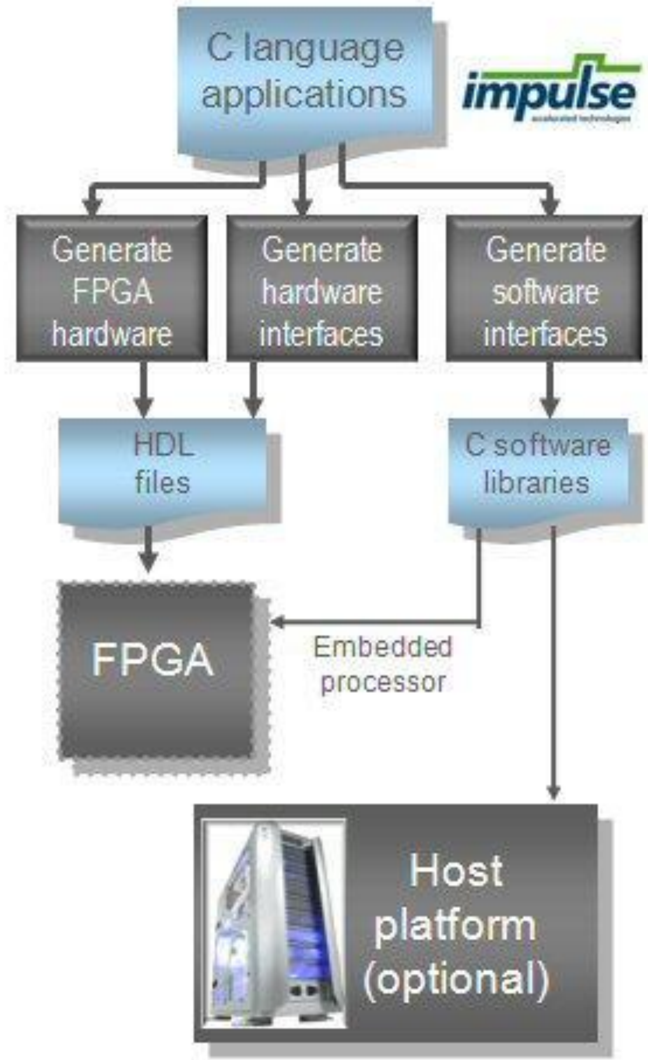
```
while (readStream(&nSample))
{
        nSample = ~nSample;
        writeStream(nSample);
}
```

Two Goals

- Improve statement-level parallelism
  - Number of code stages
  - Memory usage
  - Stage pipeline
  - Loop manipulation
- Improve system-level parallelism
  - Separation of application algorithms into multiple processes
  - Application-level pipeline through modified design
  - Reduce software processes and increase hardware processes

Design Flow

- Two separate methods for hardware and software compilation
- Hardware
  - RTL transformations
  - Multi-pass compiler stages
- Software
  - Standard compilation into machine language (usually done with GCC)
- Our focus is on hardware compilation and optimization



Stages of Hardware Compiler

- C pre-processing
- C analysis
- Initial optimization
- Loop unrolling
- Secondary optimization
- Hardware generation

C pre-processing

- Similar to its software counterpart
  - File references are resolved
  - Macro expansions are performed

C analysis

- Hardware and software processes are identified
  - **co_process_config** function is examined to determine exactly which processes are hardware (located within **co_initialize**)
- Components (e.g. communication interfaces) and their configurations are identified and analyzed for hardware/software interfaces

Initial optimization

- Basic compiler optimization techniques
- Constant folding
  - Simplifying constant expressions at compile time
  - X = 1 + 2 + 3 + 5 -> X = 11
- Dead code elimination
  - Reduces the size of logic resources used
- Certain compiler pre-optimizations in support of later passes are also performed
  - Details kept under the hood

Loop unrolling

- **UNROLL** pragma usage is found and expanded into equivalent parallel statements
- Generates errors and warnings when certain loops flagged with the pragma cannot be unrolled
  - One with non-static termination count

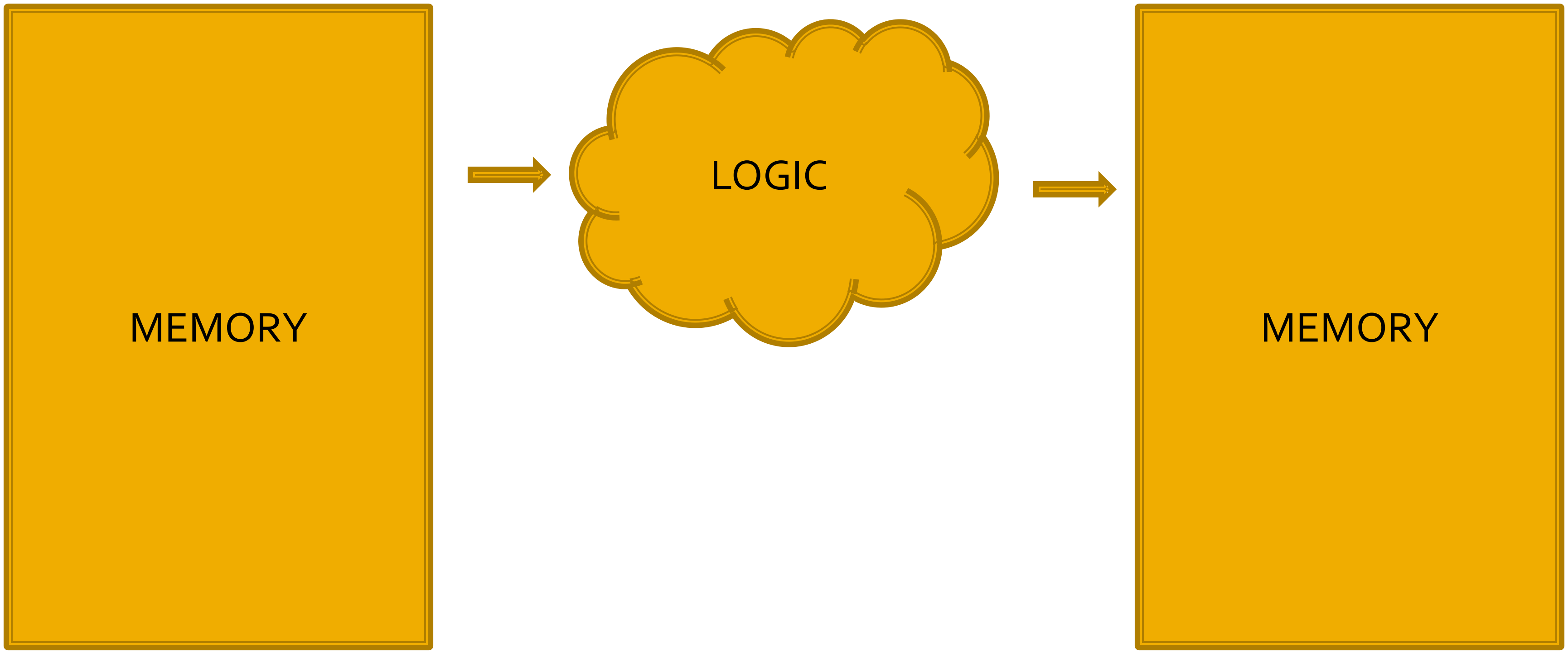
Secondary optimization

- Also referred to as the instruction stage optimization
- Key optimizations are made to extract parallelism at the level of individual C statements and at the level of code blocks
  - Condensing two or more sequential statements into one that can be executed in a single cycle
  - Re-arranging memory access statements
- **PIPELINE** pragma usage is identified and applied
  - Warnings and errors similar to loop unrolling will be generated if the loop cannot be pipelined

Hardware generation

- Code is translated into equivalent (RTL) hardware descriptions
  - Builds synthesizable HDL files for use on FPGAs
  - Drop the files right into Xilinx or Altera tools to build hardware and download to chip

Ideal Design



Impulse C Limitations

- Our optimizations will only get us so far towards this design
- As mentioned earlier, Impulse C provides a “good enough” alternative to writing pure HDL
- Tradeoff between application development time and desired performance and size

Four Key Optimizations

- Improving stream performance
- Modifying memory usage
  - Array splitting
- Loop manipulation
- Pipelining (design and code-level)

Improving stream performance

- The width and depth of a stream have a direct impact on its performance and hardware size
  - A deeper stream requires more hardware to implement
  - A wider stream also requires more hardware, but it may also make better use of the communication bus available on the target platform
- The most efficient streams use data widths that match the bus widths and depths that are just large enough to fit all possible data

Modifying memory usage

- A single memory access requires at least one clock cycle
  - Balance the width of array data by taking the target platform limitations into consideration
- Better yet, remove arrays altogether and use single variables
  - In hardware, these values are stored in "registers" that can easily be fetched/modified in parallel with other operations
- Global versus local arrays
  - Processes have access to all ports of the array's memory if put into local scope, thus increasing the number of accesses per cycle

Array splitting

- Hardware applications are capable of accessing multiple memory resources in a single clock cycle
- Break up large arrays into multiple, smaller arrays that can be accessed in parallel.

```
co_uint32 i, A[4], B[4], C[4];

for (i = 0; i < 4; i++) {
    A[i] = B[i] + C[i];
}
```

Loop manipulation

- Loop usage directly impacts application performance and the amount of logic resources used
  - Unroll loops to create an equivalent series of statements that can be executed in parallel
  - Roll loops to reduce code size and amount of logic resources used
- Unrolling is an application- and context-specific technique

Pipelining

- Another one of the most significant types of optimizations to consider for applications
- Allows loops to be performed in "parallel" with each other within processes
- Allows processes to be chained together to provide system-level (temporal) parallelism