Porting ESCAPE Cloud Microphysics Dwarf to an FPGA

J. Targett, M. Lange, O. Marsden

13 June 2019
Moore’s Law is ending - what now?

Numerical Weather Prediction has relied on Moore’s Law for quite some time

What will HPC systems look like once reality kicks in?

- How do we program it? Can we use it efficiently?
- And how many power stations do I need to run that machine?
EuroEXA: Building an FPGA+ARM exascale prototype

The EuroEXA project

▶ Co-design a novel HPC architecture capable of scaling peak performance to the exascale

▶ Cost-efficient, modular design, ARM CPUs + FPGA accelerators, unique switching interconnect, memory compression and unified memory

▶ Provide a homogenised software platform to port and optimize a rich mix of HPC applications

ECMWF’s strategy

▶ Port IFS RAPS benchmark to ARM cores

▶ Evaluate feasibility of porting large Fortran kernels to FPGAs

▶ Explore different FPGA programming models (MaxJ, OmpSS, OpenCL, ...)

▶ Explore automated source-to-source translation to aid porting efforts
Exploring programming models for NWP software

ESCAPE / ESCAPE-2 projects

▶ Break NWP software down to algorithmic building blocks, or *dwarfs*

▶ Optimise / investigate alternative formulations for each dwarf in isolation

▶ Cloud microphysics chosen as one of the NWP algorithmic components, representative of physical parameterisations

▶ Down the line, reassemble dwarfs → improved NWP code base
CLOUDSC: Cloud microphysics parameterization

- Models tendencies of five prognostic variables (cloud liquid, cloud ice, rain, snow and humidity) and the different fluxes between them
- Presence of non-linear terms.
- Terms are treated either explicitly in time (slow terms) or implicitly (fast terms).
- Takes up approx. 10% of operational runtime
Tendency for variable $\alpha$:

$$tend_{\alpha} = \frac{\partial q_{\alpha}}{\partial t} = A_{\alpha} + \frac{1}{\rho} \frac{\partial \rho V_{\alpha} q_{\alpha}}{\partial z}$$

Solved by vertical FD, implicit in time. This leads to main body of routine being (very long) vertical loop, which

- computes explicit matrix terms, and resulting tendency contribution
- computes implicit matrix terms, including one with vertical (*loop-carried*) dependency
- LU solves implicit contribution

Routine contains more than 250 local variables, more than 2500 lines in main vertical loop.
Automated source-to-source transformation

**Loki**\(^1\): Python tool to rapidly develop source-to-source transformations

- Multiple frontends to parse Fortran into AST: OFP/Rose, OMNI, FParser (WIP)
- Exposes internal high-level AST and tree visitors (language-agnostic)
- Generating high level of abstraction from low-level code requires expert input
- Pythonic “code-your-own” interface allows customization for individual models

**The aim:** Exploring alternative programming models through source-to-source translation

- Source-to-source conversion allows re-capturing some(!) high-level abstractions from regularized low-level code patterns (can be guided by expert user)
- Experimental adaptation to alternative programming paradigms and toolchains

**The plan:** Convert complex Fortran kernel and data structures to C

- Enable manual experimentation with dataflow programming, eg. MaxJ
- Eventually integrate expert knowledge to automate conversion (as far as possible)

---

\(^1\) Loki is a Norse trickster god that often uses shape shifting (transformations) to cause mischief and chaos.
Implementation - Iteration Space

Inputs: 46 floats/cell, 2 floats + 2 bools/column
Outputs: 24 floats/cell, 1 float/column
Implementation - Iteration Space

Dependencies: 32 floats between cells intra-column
Implementation - Iteration Space

\[ \sim 1500 \text{ stage pipeline along longest dependency path} \]

Targett et al.
Porting ESCAPE Cloud Microphysics Dwarf to an FPGA
Implementation - Iteration Space

~1500 stage pipeline along longest dependency path
Implementation - Iteration Space

32 floats \times \text{millions of columns} \Rightarrow \text{huge buffer requirement}
Implementation - Iteration Space

32 floats $\times$ millions of columns $\Rightarrow$ huge buffer requirement

Targett et al. Porting ESCAPE Cloud Microphysics Dwarf to an FPGA
Implementation - Iteration Space

- Cells (138)
- Block (1500)
- Columns (millions)
Implementation - Port to MaxJ

Maxeler Simulation Toolchain

- Write MaxJ code and host C code in MaxIDE2 (Eclipse)
- Compile for simulation using MaxCompiler (~1 minute)
- Simulate using MaxelerOS-sim (~75,000x slower than real-time)
Implementation - Port to MaxJ

Stage 1 - Port to MaxJ

- Enforce that only variables belonging to the current cell are assigned
  - Reorganising initialisations
  - Adding/subtracting from some array indexes
- Comment out all C kernel code - both C and MaxJ produce same result.
- Re-add sections of C code while adding in MaxJ equivalent.
  - Use various intermediate variables to check equivalence.
  - Initialise undefined variables with deterministic random numbers
- Repeat until all code ported.
Implementation - Port to MaxJ

In hindsight - not the wisest process.

- Not all intermediate values checked
- Missing code and dependencies hide errors in ported code
- *Lots* of debugging
- Should not have commented out all the code. Instead should have invested the time in a full IO harness at each stage.
- Could be automated!
Implementation - Unrolling Loops

C code

```c
float_type x[5][5][number_of_columns];

//...
//x is initialised elsewhere

for (int c=0; c<number_of_columns; c+=1) {
    for (int i=0; i<=3; i+=1) {
        for (int j=(i+1); j<=4; j+=1) {
            x[i][j][c] /= x[i][i][c];
            for (int k=(i+1); k<=4; k+=1) {
                x[k][j][c] -= x[i][j][c]*x[k][i][c];
            }
        }
    }
}
```

MaxJ Port

```java
DFEVar[][] x = new DFEVar[5][5];

//...
//x is initialised elsewhere

for (int i=0; i<=3; i+=1) {
    for (int j=(i+1); j<=4; j+=1) {
        x[i][j] /= x[i][i];
        for (int k=(i+1); k<=4; k+=1) {
            x[k][j] -= x[i][j]*x[k][i];
        }
    }
}
```
Implementation - Unrolling Loops
Implementation - Conditionals

C code

```
//Defined earlier:
//float_type x[number_of_columns];

for (int c=0; c<number_of_columns; c+=1) {
    bool condition = <some calculation>;
    if (condition) {
        float_type y = <some calculation>;
        x[c] = x[c] + y;
    }
}
```

MaxJ Port

```
//Defined earlier:
//DFEVar x;

{ 
    DFEVar y = <some calculation>;
    DFEVar condition = <some calculation>;
    x = condition ? x + y : x;
}
```
Implementation - Conditionals

C code

```c
// Defined earlier:
// float_type x[number_of_columns];

for (int c=0; c<number_of_columns; c+=1) {
    bool condition = <some calculation>;
    if (condition) {
        float_type y = <some calculation>;
        x[c] = x[c] + y;
    }
}
```

MaxJ Port

```maxj
// Defined earlier:
// DFEVar x;

{  
    DFEVar y = <some calculation>;
    DFEVar condition = <some calculation>;
    x = condition ? x + y : x;
}
```

▶ "inKBounds"
Implementation - Conditionals

C code

```c
//Defined earlier:
//float_type x[number_of_columns];

for (int c=0; c<number_of_columns; c+=1) {
    bool condition = <some calculation>;
    if (condition) {
        float_type y = <some calculation>;
        x[c] = x[c] + y;
    }
}
```

MaxJ Port

```maxj
//Defined earlier:
//DFEVar x;

{ 
    DFEVar y = <some calculation>;
    DFEVar condition = <some calculation>;
    x = condition ? x + y : x;
}
```

- "inKBounds"
- Nested conditionals
Implementation - Conditionals

C code

//Defined earlier:
//float_type x1[number_of_columns];
//float_type x2[number_of_columns];

for (int c=0; c<number_of_columns; c+=1) {
  bool condition1 = <some calculation>;
  if (condition1) {
    float_type y = <some calculation>;
    bool condition2 = <some calculation>;
    if (condition2) {
      x1[c] = x1[c] + y;
    } else {
      x2[c] = x2[c] + y;
    }
  }
}

MaxJ Port

//Defined earlier:
//DFEVar x1;
//DFEVar x2;

{
  DFEVar y = <some calculation>;
  DFEVar condition1 = <some calculation>;
  DFEVar condition2 = <some calculation>;
  x1 = (condition1 & condition2)
    ? x1 : (x1 + y);
  x2 = (condition1 & !condition2)
    ? x2 : (x2 + y);
}
Implementation - Problems with \textit{log} and \textit{exp}

- Maxeler library has slightly different results to libm.
- \textit{log} relative error: Average $= 2.53 \times 10^{-13}$, largest $= 4.93 \times 10^{-17}$
- \textit{exp} relative error: Average $= 1.04 \times 10^{-14}$, largest $= 3.75 \times 10^{-16}$
- For validation - port Maxeler’s \textit{log} and \textit{exp} to C
- Bit identical
Implementation - Fitting on the chip

Stage 2 - Running in hardware

- Double precision did not fit
- Move to single precision
- exp and log still in double precision
Implementation - Data Movement

VU9P

DIMM1
input = 15 GB/s

DIMM2
input = 10 GB/s
output = 5 GB/s

DIMM3
input = 5 GB/s
output = 10 GB/s

PCle
input = 14 MB/s

Max5 DFE

Main Memory

Host

input = 14 MB/s

Max5 DFE
Performance model

If IO Bound:

\[
\frac{15 \text{ GB}}{\text{second}} \div \left( \frac{138 \text{ cells}}{\text{column}} \times \frac{24 \text{ floats}}{\text{cell}} \times \frac{4 \text{ bytes}}{\text{float}} \right) = 1.13 \text{ million columns/second}
\]

If CPU bound:

\[
\frac{C \text{ cycles}}{\text{second}} \div \frac{138 \text{ cells}}{\text{column}} = \frac{C}{138} \text{ columns/second}
\]

Set equations equal → target clock rate of 156MHz
Preliminary results

Hardware resource usage on VU9P (single precision):
- 71.9% of logic resources, 53.1% of DSP blocks
  - 43.7% of BRAM18 memory blocks and 17.1% of URAM memory blocks

Performance
- On-board memory bandwidth limit (no PCIe): 1.13 million columns/s
- Dataflow kernel compiled at 156MHz
- 156 million cells/s, equivalent to 1.07 million columns/s
- Average flops / column estimated on CPU
- \( \rightarrow \) extrapolated equivalent FPGA performance of 133.6 Gflops/s
- CPU reference run on 12-core 2.6GHz Intel Haswell, single socket is about 21 Gflops/s, but with double precision!
- Dynamic power usage is < 30W compared to ~ 95W single socket CPU (Haswell)
Discussion and Future Work

**Ported complex cloud microphysics kernel onto FPGA!**
- Converted complex Fortran code and data structures to C via source-to-source translation
- Hand-ported to MaxJ via Maxeler IDE and emulator
- Promising early performance numbers for single-precision kernel
- Performance is limited by IO bandwidth

**Future work:**
- More in-depth performance and power analysis and comparison to CPU/GPU

**Open questions:**
- Can we automate the C → MaxJ conversion?
- Can we feed data into the FPGA at an even higher rate?
- How do we handle multiple (many) of those kernels?
- What precision is really needed and where? Can we leverage true mixed precision?
This work has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 754337.