FPGA Acceleration of the LFRic Weather and Climate Model in the EuroExa Project Using Vivado HLS

Mike Ashworth, Graham Riley, Andrew Attwood and John Mawer
Advanced Processor Technologies Group
School of Computer Science,
University of Manchester, United Kingdom
mike.ashworth.compsci@manchester.ac.uk
Project outline

Horizon 2020 FETHPC-01-2016:
Co-design of HPC systems and applications
EuroExa started 1st Sep 2017, runs for 3½ years
16 Partners, 8 countries, €20M
Builds on previous projects, esp. ExaNoDe, ExaNeSt, EcoScale

Aim: design, build, test and evaluate an Exascale prototype system
Architecture based on ARM CPUs with FPGA accelerators
Three testbed systems: #3 >100 Pflop/s
Low-power design goal to target realistic Exascale system
Architecture evolves in response to application requirements = co-design
Wide range of apps, incl. weather forecasting, lattice Boltzmann, multiphysics, astrophysics, astronomy data processing, quantum chemistry, life sciences and bioinformatics

@euroexa
euroexa.eu

Kick-off meeting 4th-5th Sep 2017, Barcelona
Motivation

• FPGAs offer large (OsOM) gains in performance/W

• Also gains in performance/$£€ß$

• Major corporations are using FPGAs in datacentres for cloud services, analytics, communication, etc.

• H/W traditionally led by Xilinx (ARM CPU + FPGA single chip)

• Intel’s acquisition of Altera led to Heterogeneous Architecture Research Platform (HARP) (also single chip)

• Predictions: up to 30% of datacenter servers will have FPGAs by 2020
Brand new weather and climate model: LFRic
named after Lewis Fry Richardson (1881-1953)

- Dynamics from the GungHo project 2011-2015
- Scalability – globally uniform grid (no poles)
- Speed – maintain performance at high & low resolution and for high & low core counts
- Accuracy – need to maintain standing of the model
- Separation of Concerns – PSyClone generated layer for automated targeting of architectures
- Operational weather forecasts around 2022 – anniversary of Richardson (1922)

GLOBALLY UNIFORM NEXT GENERATION HIGHLY OPTIMIZED

“Working together harmoniously”
• Baroclinic performance benchmark case
• gprof ... | gprof2dot.py | dot ...

• Two subroutines in the Helmholtz solver use 54% of runtime
• Most is in matrix-vector products within a loop over vertical levels
Zynq UltraScale+ ZCU102 Evaluation Platform

- ARM Cortex A53 quad-core CPU 1.2 GHz
- Dual-core Cortex-R5 real-time processor
- Mali-400 MP2 GPU
- Zynq UltraScale XCZU9EG-2FFVB1156 FPGA

<table>
<thead>
<tr>
<th>System Logic Cells (K)</th>
<th>600</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory (Mb)</td>
<td>32.1</td>
</tr>
<tr>
<td>DSP Slices</td>
<td>2,520</td>
</tr>
<tr>
<td>Maximum I/O Pins</td>
<td>328</td>
</tr>
</tbody>
</table>
Approach

• LFRic
  • Full application (Fortran)
  • Compact applications “mini-apps”
  • Kernels e.g. matrix-vector product (C)

• Performance on ARM CPU quad-core A53

• Use Vivado HLS to generate IP blocks for key kernels to run on the UltraScale+ FPGA

• Investigate performance optimizations
Range of Programming Models

1. C code with Xilinx Vivado HLS and Vivado Design Suite
2. OmpSs@FPGA directive-based (BSC)
3. MaxJ compiler for Maxeler systems
4. OpenCL code with Xilinx SDSoC
5. OpenStream (Uni Man)

• Options 2-5 being investigated by other members of the project
#define NDF1 8
#define NDF2 6
#define NK 40
#define MVTYPE double

int matvec_8x6x40_vanilla (MVTYPE matrix[NK][NDF2][NDF1],
   MVTYPE x[NDF2][NK], MVTYPE lhs[NDF1][NK]) {
   int df,j,k;
   for (k=0;k<NK;k++) {
      for (df=0;df<NDF1;df++) {
         lhs[df][k] = 0.0;
         for (j=0;j<NDF2;j++) {
            lhs[df][k] = lhs[df][k] + x[j][k]*matrix[k][j][df];
         }
      }
   }
   return 0;
}

Notes:
- Data sizes hard-wired for HLS
- Vertical loop k is outer
- Vectors x and lhs are sequential in k (k-last in C)
- Matrix is not (k-first)
- Read-then-write dependence on lhs
- Flops = 2*NK*NDF1*NDF2 = 3840
- Mem refs = 2*flops = 7680 doubles
Optimizations in Vivado HLS

• Make k the inner loop (loop length 40, independent, sequential access)
• Transpose matrix to k-last to ensure sequential memory access
• HLS pragma to unroll inner loops on k (no benefit from hand unrolling)
• HLS pragma to pipeline outer loop on df
• HLS pragma for input and output arguments including
  • num_read_outstanding=8
  • max_read_burst_length=64
• Access input and output arguments by memcpy to local arrays to ensure streaming of loads/stores to/from BRAM (see later)
#pragma HLS INTERFACE m_axi depth=128
port=matrix offset=slave bundle=bram /
       num_read_outstanding=8 /
       num_write_outstanding=8 /
       max_read_burst_length=64 /
       max_write_burst_length=64
< pragmas for m_axi interfaces for x, lhs and s_axilite interface for return>

int df,j,k;
MVTYPE ml[NDF2][NK], xl[NDF2][NK],
ll[NDF1][NK];
memcpy (xl, x, NDF2*NK*sizeof(MVTYPE));
for (df=0;df<NDF1;df++) {
  #pragma HLS PIPELINE
  for (k=0;k<NK;k++) {
    #pragma HLS UNROLL
    ll[df][k] = 0.0;
  }
  memcpy (ml, matrix+df*NDF2*NK,
          NDF2*NK*sizeof(MVTYPE));
  for (j=0;j<NDF2;j++) {
    for (k=0;k<NK;k++) {
      #pragma HLS UNROLL
      ll[df][k] = ll[df][k]
                 + xl[j][k]*ml[j][k];
    }
  }
}
memcpy (lhs, ll,
        NDF1*NK*sizeof(MVTYPE));
Performance Estimate:
- Target 2ns clock: design validated at 2.89ns = 346 MHz
- 2334 cycles for 3840 flops = 1.65 flops/cycle
- Overlapped dmul with dadd
- Starting code was 69841 cycles

Utilization Estimate:
- Try to maximize performance while minimizing utilization
- Shows percentage of chip ‘real-estate being utilized

<table>
<thead>
<tr>
<th>Name</th>
<th>BRAM_18K</th>
<th>DSP48E</th>
<th>FF</th>
<th>LUT</th>
<th>URAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Expression</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>701</td>
<td>-</td>
</tr>
<tr>
<td>FIFO</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Instance</td>
<td>4</td>
<td>10</td>
<td>2527</td>
<td>2222</td>
<td>-</td>
</tr>
<tr>
<td>Memory</td>
<td>4</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>Multiplexer</td>
<td>-</td>
<td>-</td>
<td>4280</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Register</td>
<td>-</td>
<td>-</td>
<td>20672</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Total</td>
<td>8</td>
<td>10</td>
<td>23199</td>
<td>7203</td>
<td>-</td>
</tr>
<tr>
<td>Available</td>
<td>1824</td>
<td>2520</td>
<td>548160</td>
<td>274080</td>
<td>0</td>
</tr>
<tr>
<td>Utilization (%)</td>
<td>~0</td>
<td>~0</td>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
• Design built from the following IP blocks:
  • 12x matrix-vector multiply IP blocks from HLS
  • 12x Block Memory Generators for BRAM memory
  • 12x AXI BRAM Controller
  • 14x AXI Crossbar blocks for switching
  • 1x AXI Interconnect blocks for switching
  • 1x AXI Protocol Converter
  • 1x ZynQ PS block for interface with ARM
  • 1x Clocking Wizard to multiply system clock
  • 1x Processor System Reset

• Connect blocks manually or with automated assistance
• Address mapping manually or automated using TCL scripts
  • Address editor determines size of BRAM blocks (12x 256kB)
• Vivado SDK reveals “register” assignment for HLS blocks
Design with 12 Matrix-Vector Blocks
Notes:

- "Reg" location in memory of control words for each matrix-vector IP block (8kB)
- "Mem0" location of BRAM memory block (256kB)
Vivado SDK Registers

Notes:

• Not hardware “registers”, these are locations in memory

• To execute the block:
  - Load addresses of arrays in BRAM into the “registers” for matrix, x and lhs
  - Set AP_START bit in CTRL word to “1”
  - Check for AP_IDLE bit returning to “1” to indicate completion
Optimized Data Streaming - Before

Produced by attaching an Integrated Logic Analyzer IP block to the data path going into the matrix-vector IP block

One read at a time
Optimized Data Streaming - After

Burst at one read per cycle
### Notes:
- Using most of the BRAM memory
- Using only 7% of DSPs
- Using around half the other logic (LUT+FF)
ARM driver code

• Setup a two devices /dev/uio0 and /dev/uio1 – two ports on the ZynQ block
• Use mmap to map the FPGA memory into user space
• Assign pointers for each data array to location in user space
• Control loop to divide up the work into 12 “chunks” which will fit into the FPGA BRAM memory (maximum 12 x 256kB = 3MB) (13 columns in this LFRic model)
• For each chunk:
  • Assign work to one of the matrix-vector blocks
  • Copy input data into BRAM
  • Set the control word “registers” for the block
  • Start the block by setting AP_START
  • Wait for block to finish by watching AP_IDLE (opportunity for overlap)
  • Copy output data from BRAM
• In practice we fill 3MB BRAM, then run all 12 matrix-vector blocks, then copy output data back and repeat
• Check correctness and time the code
Results for 12 blocks

- 510 Mflop/s per block => 1.53 flops/cycle (93% of HLS estimate)
- Parallel efficiency at 12 IP blocks 87%
- Clock scaling 100 to 333 MHz is 94% efficient
- ARM Cortex A53 single core 227 Mflop/s – SAME CODE
- ARM quad-core with OpenMP 800 Mflop/s approx.
- FPGA:ARM quad-core speed-up: 6.7x
Critical Performance Factors

- Clock speed
- Number of matrix-vector blocks
- Performance of single matrix-vector block
Summary

We have

• Used Vivado HLS to develop a matrix-vector kernel which runs on the UltraScale+ FPGA at 5.3 double precision Gflop/s (single precision: similar performance, 63% resources)

Issues

• Timing constraints in the Vivado design prevent larger numbers of blocks and higher clock speeds
• Therefore cannot exploit all the FPGA logic for this algorithm
Future work

• Generate an IP block and driver for the LFRic code: apply_variable_hx_kernel_code (HLS done; 1.75 flops/cycle)

• Exploit MPI within LFRic to run across multiple nodes and multiple FPGAs (done trivially with the matrix-vector kernel)

• How many other kernels can we port to the FPGAs?

• Can we link kernels to avoid data transfer?

• When do we need to reconfigure? At what cost?

• Future hardware: now ZU9, VU9 (early 2019) and HBM (Xilinx white paper)
Many thanks
Please connect at @euroexa or euroexa.eu

Mike Ashworth, Graham Riley, Andrew Attwood and John Mawer
Advanced Processor Technologies Group
School of Computer Science,
University of Manchester, United Kingdom
mike.ashworth.compsci@manchester.ac.uk