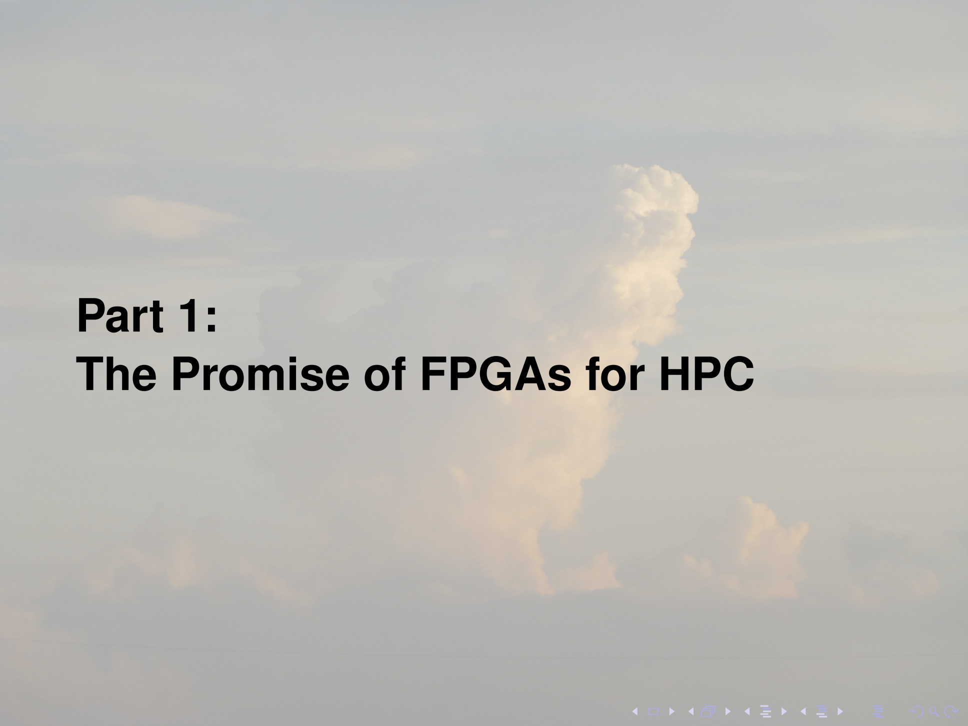
"On the Capability and Achievable Performance of FPGAs for HPC Applications"

Wim Vanderbauwhede
School of Computing Science, University of Glasgow, UK

Or in other words

"How Fast Can Those FPGA Thingies Really Go?"

Outline



Part 1: The Promise of FPGAs for HPC

High-end FPGA Board

64GB RAM, 32GB/s

30 Watt

8-lane PCIe

4 FPGAs with
3M logic elements

FPGAs in a Nutshell

Field-Programmable Gate Array

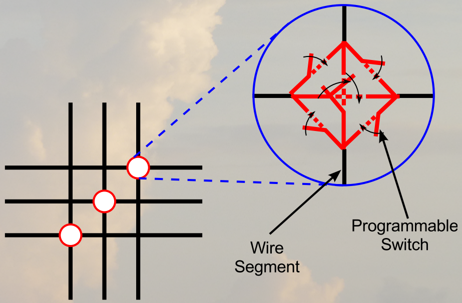
- Configurable logic
- Matrix of look-up tables (LUTs) that can be configured into any N-input logic operation
- e.g. 2-bit LUT configured as XOR:

Address	Value
00	1
01	0
10	0
00	1

- Combined with flip-flops to provide state

FPGAs in a Nutshell

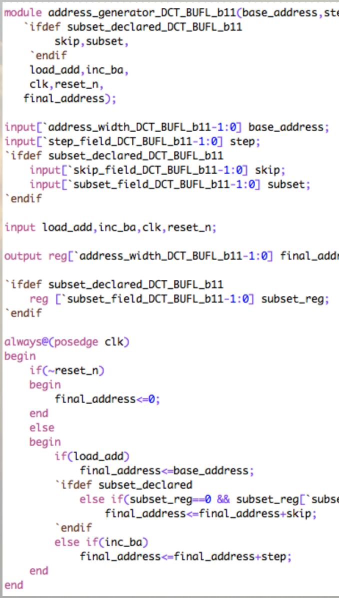
- Communication fabric:
  - island-style: grid of wires with islands of LUTS
  - wires with switch boxes
  - provides full connectivity
- Also dedicated on-chip memory



FPGAs in a Nutshell

Programming

- By configuring the LUTs and their interconnection, one can create arbitrary circuits
- In practice, circuit description is written in VHDL or Verilog, and converted into a configuration file by the vendor tools
    - Two major vendors: Xilinx and Altera
- Many "C-based" programming solutions have been proposed and are commercially available. They generate VHDL or Verilog.
- Most recently, OpenCL is available for FPGA programming (specific Altera-based boards only)



```
module address_generator_DCT_BUFL_b11(base_address,ste
   `ifdef subset_declared_DCT_BUFL_b11
       skip,subset,
    `endif
    load_add,inc_ba,
    clk,reset_n,
   final_address);

input[`address_width_DCT_BUFL_b11-1:0] base_address;
input[`step_field_DCT_BUFL_b11-1:0] step;
`ifdef subset_declared_DCT_BUFL_b11
    input[`skip_field_DCT_BUFL_b11-1:0] skip;
    input[`subset_field_DCT_BUFL_b11-1:0] subset;
`endif

input load_add,inc_ba,clk,reset_n;

output reg[`address_width_DCT_BUFL_b11-1:0] final_addr

`ifdef subset_declared_DCT_BUFL_b11
    reg [`subset_field_DCT_BUFL_b11-1:0] subset_reg;
`endif

always@(posedge clk)
begin
    if(~reset_n)
    begin
        final_address<=0;
    end
    else
    begin
        if(load_add)
            final_address<=base_address;
        `ifdef subset_declared
            else if(subset_reg==0 && subset_reg[`subse
                final_address<=final_address+skip;
        `endif
        else if(inc_ba)
            final_address<=final_address+step;
    end
end
```

The Promise

FPGAs have great potential for HPC:

- Low power consumption
- Massive amount of fine grained parallelism (e.g. Xilinx Virtex-6 has about 600,000 LUTs)
- Huge (TB/s) internal memory bandwidth
- Very high power efficiency (GFLOPS/W)

The Challenge

FPGA Computing Challenge

- Device clock speed is very low
- Many times lower than memory clock
- There is no cache
- So random memory access will kill the performance
- Requires a very different programming paradigm
- So, it's hard
- But that shouldn't stop us

Maximum Achievable Performance

The theoretical maximum computational performance is determined by:

- Available Memory bandwidth
  - Easy: read the datasheets!
- Compute Capacity
  - Hard: what is the relationship between logic gates and “FLOPS”?

FLOPS?

What is a FLOP, anyway?

- Ubiquitous measure of performance for HPC systems
- Floating-point Operations per second
- Floating-point:
  - Single or double precisions?
  - Number format: floating-point or fixed-point?
- Operations:
  - Which operations? Addition? Multiplication
  - In fact, why *floating point*?
  - Depends on the application

FLOPS!

FLOPS on Multicore CPUs and GPGPUs

- Fixed number of FPUs
- Historically, FP operations had higher cost than integer operations
- Today, essentially no difference between integer and floating-point operations
- But scientific applications perform mostly FP operations
- Hence, FLOPS as a measure of performance

An Aside: the GPGPU Promise

- Many papers report huge speed-ups: 20x/50x/100x/...
- And the vendors promise the world
- However, theatrical FLOPS are comparable between same-complexity CPUs and GPGPUs:

	#cores	vector size	Clock speed (GHz)	GFLOPS
CPU: Intel Xeon E5-2640	24	8	2.5	480
GPU: Nvidia GeForce GX480	15	32	1.4	672
CPU: AMD Opteron 6176 SE	48	4	2.3	442
GPU: Nvidia Tesla C2070	14	32	1.1	493
FPGA: GiDEL PROCStar-IV	*?*	*?*	*0.2*	*??*

- Difference is no more than 1.5x

The GPGPU Promise (Cont’d)

- Memory bandwidth is usually higher for GPGPU:

	Memory BW (GB/s)
CPU: Intel Xeon E5-2640	42.6
GPU: Nvidia GeForce GX480	177.4
CPU: AMD Opteron 6176 SE	42.7
GPU: Nvidia Tesla C2070	144
FPGA: GiDEL PROCStar-IV	*32*

- The difference is about 4.5x
- So where do the 20x/50x/100x figures come from?
- Unoptimised baselines!

FPGA Power Efficiency Model (1)

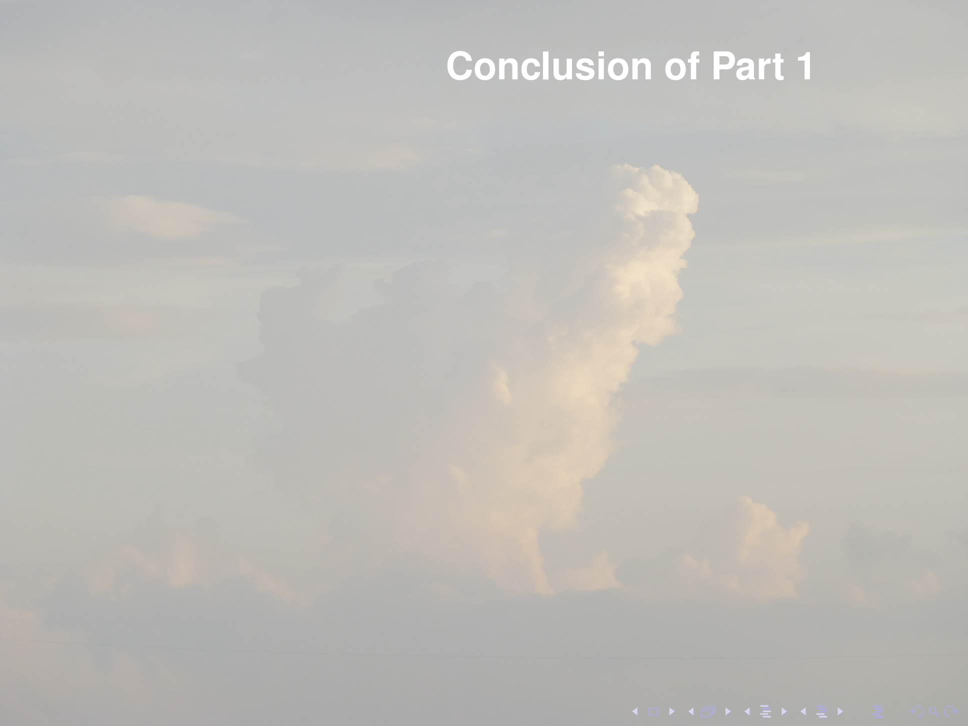
- On FPGAs, different instructions (e.g. *, +, /) consume different amount of resources (area and time)
- FLOPS should be defined on a per-application basis
  - We analyse the application code and compute the aggregated resource requirements based on the count $n_{OP,i}$ and resource utilisation $r_{OP,i}$ of the required operations
    $r_{app} = \sum_{i=1}^{N_{instrs}} n_{OP,i} r_{OP,i}$
  - We take into account an area overhead ϵ for control logic, I/O etc.
- Combined with the available resources on the board r_{FPGA}, the clock speed f_{FPGA} and the power consumption P_{FPGA}, we can compute the power efficiency:
  Power Efficiency=$(1-\epsilon)(r_{FGPA}/r_{app})/f_{FPGA}/P_{FPGA}$ GFLOPS/W

FPGA Power Efficiency Model (2)

Example: convection kernel from the FLEXPART Lagrangian particle dispersion simulator

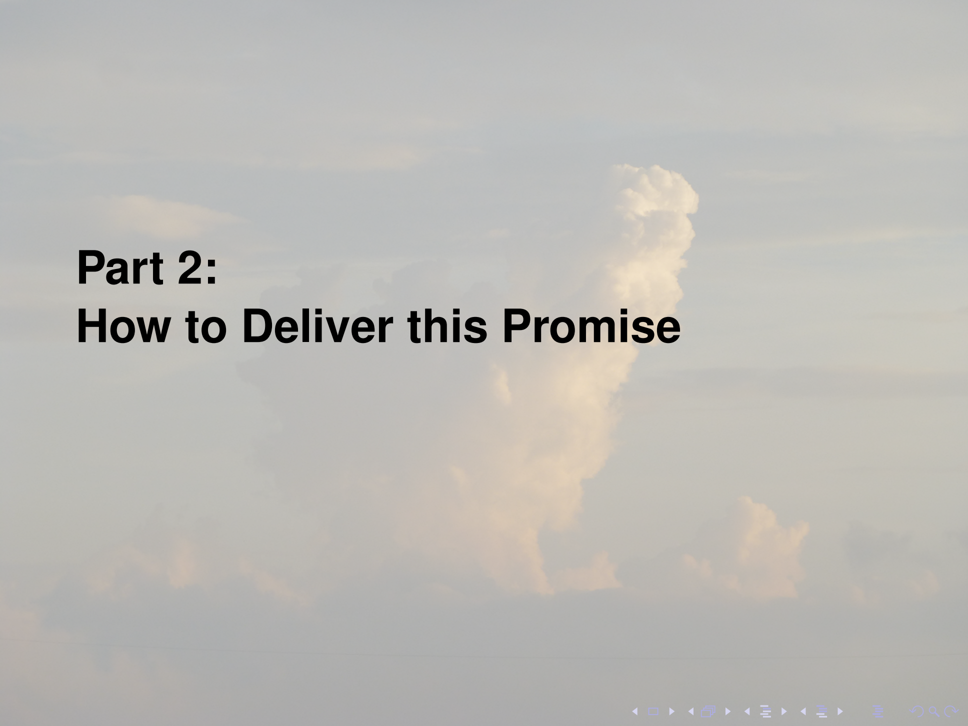
- About 600 lines of Fortran 77
- This would be a typical kernel for e.g. OpenCL or CUDA on a GPU
- Assuming a GiDEL PROCStar-IV board, $P_{FPGA} = 30W$
- Assume $\epsilon = 0.5$ (50% overhead, conservative) and clock speed $f_{FPGA} = 175MHz$ (again, conservative)
  - Resulting power efficiency: **30 GFLOPS/W**
  - By comparison: Tesla C2075 GPU: **4.5 GFLOPS/W**
  - If we only did multiplications and similar operations, it would be **15 GFLOPS/W**
  - If we only did additions and similar operations, it would be **225 GFLOPS/W**
- Depending on the application, the power efficiency can be up to **50x** better on FPGA!

Conclusion of Part 1



Conclusion of Part 1

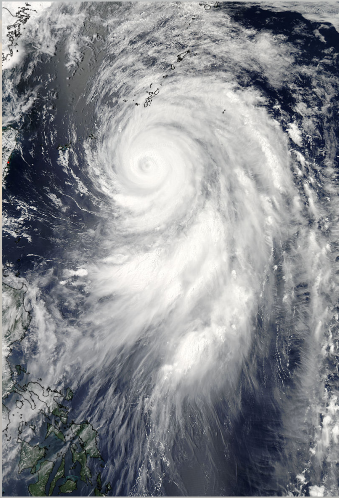
The FPGA HPC promise is real!



Part 2:
How to Deliver this Promise

Assumptions on Applications

- Suitable for streaming computation
- Data parallelism
- If it works well in OpenCL or CUDA, it will work well on FPGA
- Single-precision floating point, integer or bit-level operations. Doubles take too much space.
- Suitable model for many scientific applications (esp. NWP)
- But also for data search, filtering and classification
- So good for both HPC and data centres



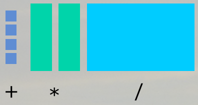
Computational Architecture

- Essentially, a network of processors
  - But "processors" defined very loosely
  - Very different from e.g. Intel CPU
    - Streaming processor
    - Minimal control flow
    - Single-instruction
    - Coarse-grained instructions
- Main challenge is the parallelisation
  - Optimise memory throughput
  - Optimise computational performance

Example

A – somewhat contrived – example to illustrate our optimisation approach:

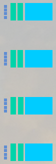
- We assume we have an application that performs 4 additions, 2 multiplications and a division
- We assume that the relative areas of the operations are 16, 400, 2000 slices
- We assume that the multiplication requires 2 clock cycles and the division requires 8 clock cycles
- The processor area would be 4*64+2*200+1*2000 = 2528 slices
- The compute time 1*4+2*2+8*1 = 16 cycles



Lanes

Memory clock is several times higher than FPGA clock: $f_{MEM} = n.f_{FPGA}$

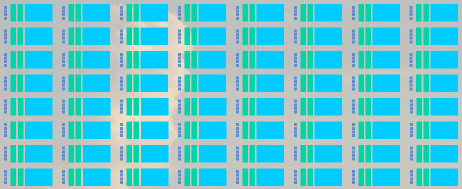
- To match memory bandwidth requires at least n parallel lanes
  - For the GiDEL board, $n = 4$
  - So the area requirement is 10,000 slices
  - But the throughput is still only 1/16th of the memory bandwidth



Threads

Typically, each lane needs to perform many operations on each item of data read from memory (16 in the example)

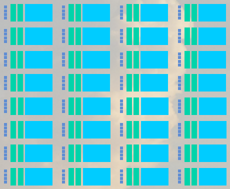
- So we need to parallelise the computational units per lane as well
- A common approach is to use data parallel threads to achieve processing at memory rate
  - In our example, this requires 16 threads, so 160,000 slices

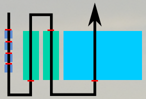


Pipelining

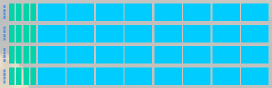
However, this approach is wasteful:

- Create a pipeline of the operations
- Each stage in the pipeline on needs the operation that it executes
    - In the example, this requires 4*16+2*400+1*2000 slices, and 8 cycles per datum
    - Requires only 8 parallel threads to achieve memory bandwidth, so 80,000 slices





Balancing the Pipeline



This is still not optimal:

- As we assume a streaming mode, we can replicate pipeline stage to balance the pipeline
- In this way, the pipeline will have optimal throughput
  - In the example, this requires 4*16+2*2*400+8*2000 slices to process at 1 cycle per datum
  - So the total resource utilisation is 17,664*4=70,656 slices
- To evaluate various trade-offs (e.g lower clock speeds/ smaller area/ more cycles), we use the notion of "Effective Slice Count" (ESC) to express the number of slices required by an operation in order to achieve a balanced pipeline.

Coarse Grained Operations

We can still do better though:

- By grouping fine-grained operations into coarser-grained ones, we reduce the overhead of the pipeline.
- This is effective as long as the clock speed does not degrade
- Again, the ESC is used to evaluate the optimal grouping

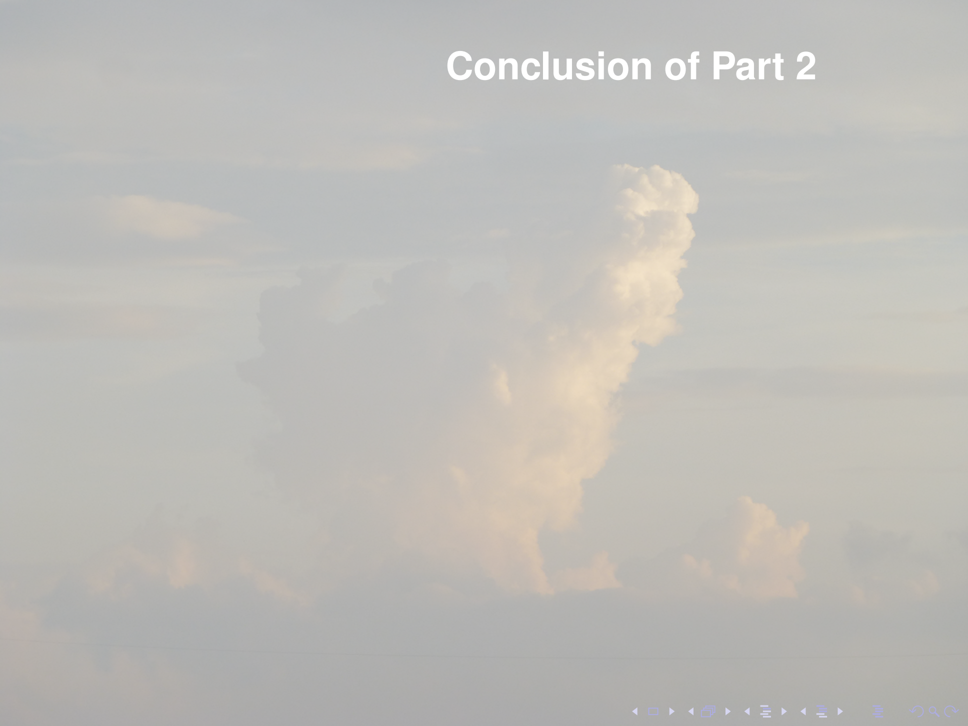
Preliminary Result

- We applied our approach manually to a small part of the convection kernel
- The balanced pipeline results in 10GFLOPS/W, without any optimisation in terms of number representation
- This is already better than a Tesla C2075 GPU

Application Size

- The approach we outlined leads to optimal performance *if the circuit fits on the FPGA*
- What if the circuit is too large for the FPGA (and you can't buy a larger one)?
  - Only solution is to trade space for time, i.e. reduce throughput
  - Our approach is to group operations into processors
  - Each processor instantiates the instruction required to perform all operations
  - Because some instructions are executed frequently, there is an optimum for operations/area
  - As the search space is small, we perform an exhaustive search for the optimal solution
- The throughput drops with the number of operations per processor, so based on the theoretical model, for our example case with 4 to 8 operations it can still be worthwhile to use the FPGA.

Conclusion of Part 2



Conclusion of Part 2

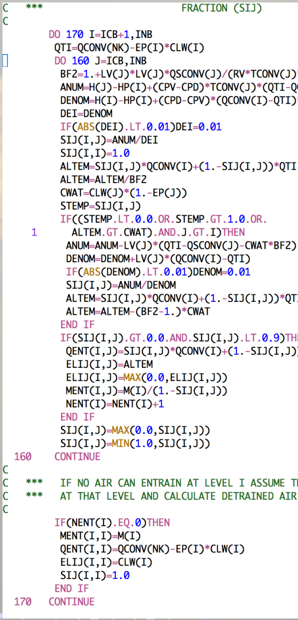
The FPGA HPC Promise can be delivered –

Conclusion of Part 2

– but it's hard work!

A Matter of Programming

- In practice, scientists don't write "**streaming multiple-lane balanced-pipeline**" code. They write code like this $\longrightarrow\longrightarrow\longrightarrow\longrightarrow\longrightarrow$
- And current high-level programming tools still require a lot of programmer know-how to get good performance, because essentially the only way is to follow a course as outlined in this talk.
- So we need better programming tools
- And specifically, better compilers



```
C     ***                            FRACTION (SIJ)
C
          DO 170 I=ICB+1,INB
           QTI=QCONV(NK)-EP(I)*CLW(I)
           DO 160 J=ICB,INB
            BF2=1.+LV(J)*LV(J)*QSCONV(J)/(RV*TCONV(J)
            ANUM=H(J)-HP(I)+(CPV-CPD)*TCONV(J)*(QTI-Q
            DENOM=H(I)-HP(I)+(CPD-CPV)*(QCONV(I)-QTI)
            DEI=DENOM
            IF(ABS(DEI).LT.0.01)DEI=0.01
            SIJ(I,J)=ANUM/DEI
            SIJ(I,I)=1.0
            ALTEM=SIJ(I,J)*QCONV(I)+(1.-SIJ(I,J))*QTI
            ALTEM=ALTEM/BF2
            CWAT=CLW(J)*(1.-EP(J))
            STEMP=SIJ(I,J)
            IF((STEMP.LT.0.0.OR.STEMP.GT.1.0.OR.
     1        ALTEM.GT.CWAT).AND.J.GT.I)THEN
             ANUM=ANUM-LV(J)*(QTI-QSCONV(J)-CWAT*BF2)
             DENOM=DENOM+LV(J)*(QCONV(I)-QTI)
             IF(ABS(DENOM).LT.0.01)DENOM=0.01
             SIJ(I,J)=ANUM/DENOM
             ALTEM=SIJ(I,J)*QCONV(I)+(1.-SIJ(I,J))*QTI
             ALTEM=ALTEM-(BF2-1.)*CWAT
            END IF
            IF(SIJ(I,J).GT.0.0.AND.SIJ(I,J).LT.0.9)TH
             QENT(I,J)=SIJ(I,J)*QCONV(I)+(1.-SIJ(I,J)
             ELIJ(I,J)=ALTEM
             ELIJ(I,J)=MAX(0.0,ELIJ(I,J))
             MENT(I,J)=M(I)/(1.-SIJ(I,J))
             NENT(I)=NENT(I)+1
            END IF
            SIJ(I,J)=MAX(0.0,SIJ(I,J))
            SIJ(I,J)=MIN(1.0,SIJ(I,J))
  160      CONTINUE
C
C     ***   IF NO AIR CAN ENTRAIN AT LEVEL I ASSUME T
C     ***   AT THAT LEVEL AND CALCULATE DETRAINED AIR
C
           IF(NENT(I).EQ.0)THEN
            MENT(I,I)=M(I)
            QENT(I,I)=QCONV(NK)-EP(I)*CLW(I)
            ELIJ(I,I)=CLW(I)
            SIJ(I,I)=1.0
           END IF
  170     CONTINUE
```

Enter the TyTra Project

- Project between universities of Glasgow, Heriot-Watt and Imperial College, funded by EPSRC
- The aim: compile scientific code efficiently for heterogeneous platforms, including multicore/manycore CPUs GPGPUs and FPGAs
- The approach: TYpe TRAnsformations
  - Infer the type of all communication in a program
  - Transform the types using a formal, provably correct mechanism
  - Use a cost model to identify the suitable transformations
- Five-year project, started Jan 2014

But Meanwhile

A practical recipe:

- Given a legacy Fortran application
- And a high-level FPGA programming solution, e.g. Maxeler, Impulse-C, Vivado or Altera OpenCL
- Rewrite your code in data-parallel fashion, e.g in OpenCL
  - There are tools to help you: automated refactoring, Fortran-to-C translation
  - This will produce code suitable for streaming
- Now rewrite this code to be similar to the pipeline model described
- Finally, rewrite the code obtained in this way for Maxeler, Impulse-C etc,mainly a matter of syntax

Conclusion

- FPGAs are very promising for HPC
- We presented a model to estimate the maximum achievable performance on a per-application basis
- Our conclusion is that the power efficiency can be up to 10x better compared to GPU/multicore CPU
- We presented a methodology to achieve the best possible performance
- Better tools are needed, but already with today’s tools very good performance is achievable

Thank you