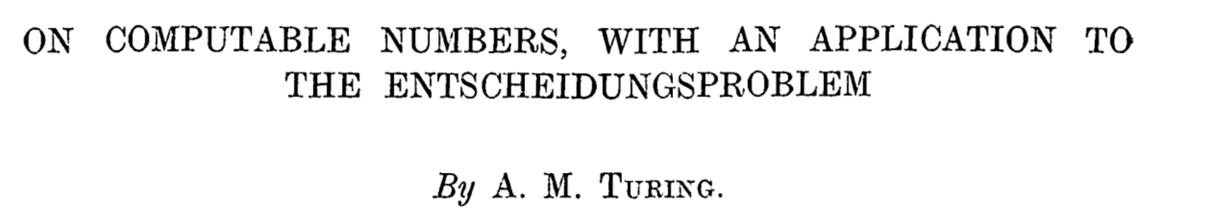
FPGAs as Components in Heterogeneous HPC Systems: Raising the Abstraction Level of Heterogeneous Programming

Wim Vanderbauwhede
School of Computing Science
University of Glasgow

A trip down memory lane

80 Years ago: The Theory



ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

By A. M. TURING.

Turing, Alan Mathison. "On computable numbers, with an application to the Entscheidungsproblem." *J. of Math* 58, no. 345-363 (1936): 5.

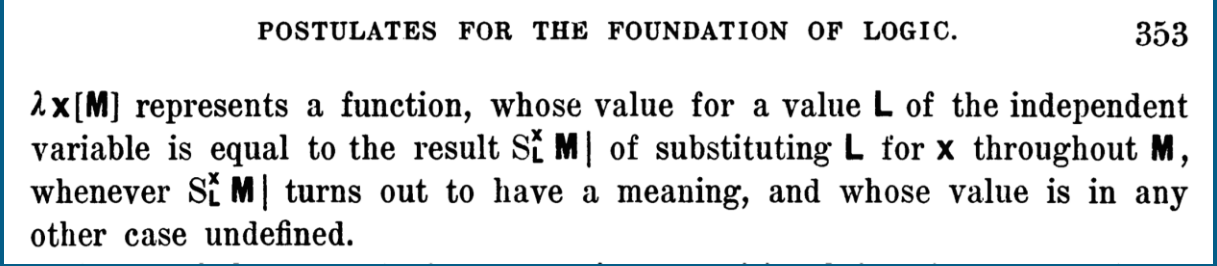
1936: Universal machine (Alan Turing)

1936: Lambda calculus (Alonzo Church)

1936: Stored-program concept (Konrad Zuse)

1937: Church-Turing thesis

1945: The Von Neumann architecture

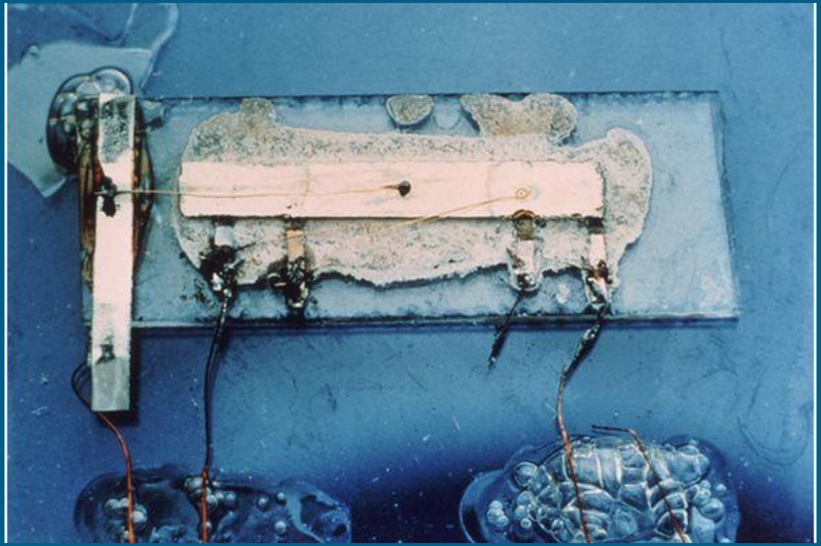


POSTULATES FOR THE FOUNDATION OF LOGIC. 353

$\lambda \mathbf{x}[\mathbf{M}]$ represents a function, whose value for a value $\mathbf{L}$ of the independent variable is equal to the result $\mathrm{S}^{\mathbf{x}}_{\mathbf{L}}\mathbf{M}|$ of substituting $\mathbf{L}$ for $\mathbf{x}$ throughout $\mathbf{M}$, whenever $\mathrm{S}^{\mathbf{x}}_{\mathbf{L}}\mathbf{M}|$ turns out to have a meaning, and whose value is in any other case undefined.

Church, Alonzo. "A set of postulates for the foundation of logic." *Annals of mathematics* (1932): 346-366.

60-40 Years ago: The Foundations



The first working integrated circuit, 1958. © Texas Instruments.

1957: Fortran, John Backus, IBM

1958: First IC, Jack Kilby, Texas Instruments

1965: Moore’s law

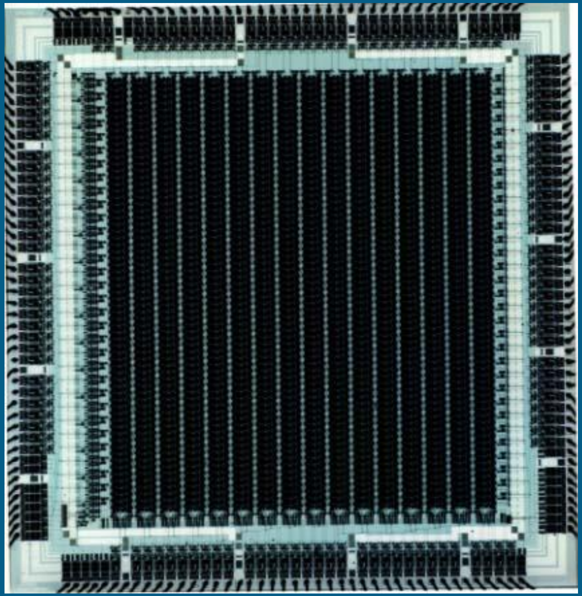
1971: First microprocessor, Texas Instruments

1972: C, Dennis Ritchie, Bell Labs

1977: Fortran-77

1977: von Neumann bottleneck, John Backus

30 Years ago:
HDLs and FPGAs



Algotronix CAL1024 FPGA, 1989. © Algotronix

1984: Verilog

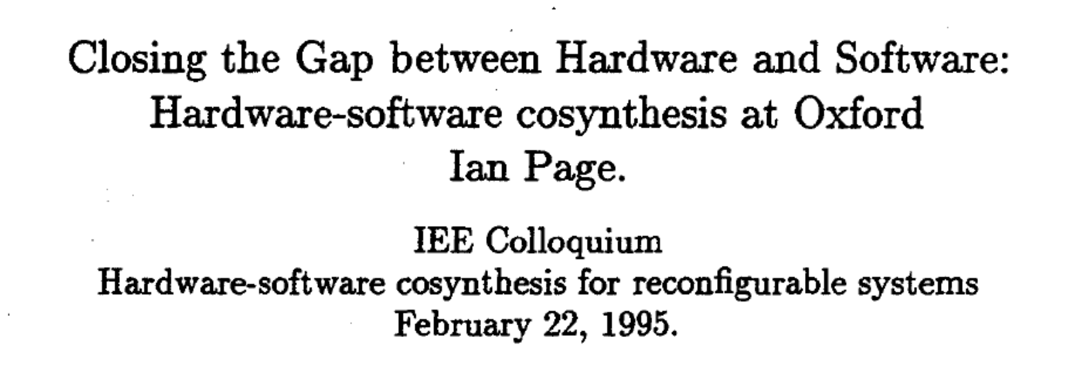
1984: First reprogrammable logic device, Altera

1985: First FPGA,Xilinx

1987: VHDL Standard IEEE 1076-1987

1989: Algotronix CAL1024, the first FPGA to offer random access to its control memory

20 Years ago: High-level Synthesis



Closing the Gap between Hardware and Software: Hardware-software cosynthesis at Oxford

Ian Page.

IEE Colloquium

Hardware-software cosynthesis for reconfigurable systems

February 22, 1995.

Page, Ian. "Closing the gap between hardware and software: hardware-software cosynthesis at Oxford." (1996): 2-2.

1996: Handel-C, Oxford University

2001: Mitrion-C, Mitrionics

2003: Bluespec, MIT

2003: MaxJ, Maxeler Technologies

2003: Impulse-C, Impulse Accelerated Technologies

2004: Catapult C, Mentor Graphics

2005: SystemVerilog, BSV

2006: AutoPilot, AutoESL (Vivado)

2007: DIME-C, Nallatech

2011: LegUp, University of Toronto

2014: Catapult, Microsoft

10 Years ago: Heterogeneous Computing

2007: CUDA, NVIDIA

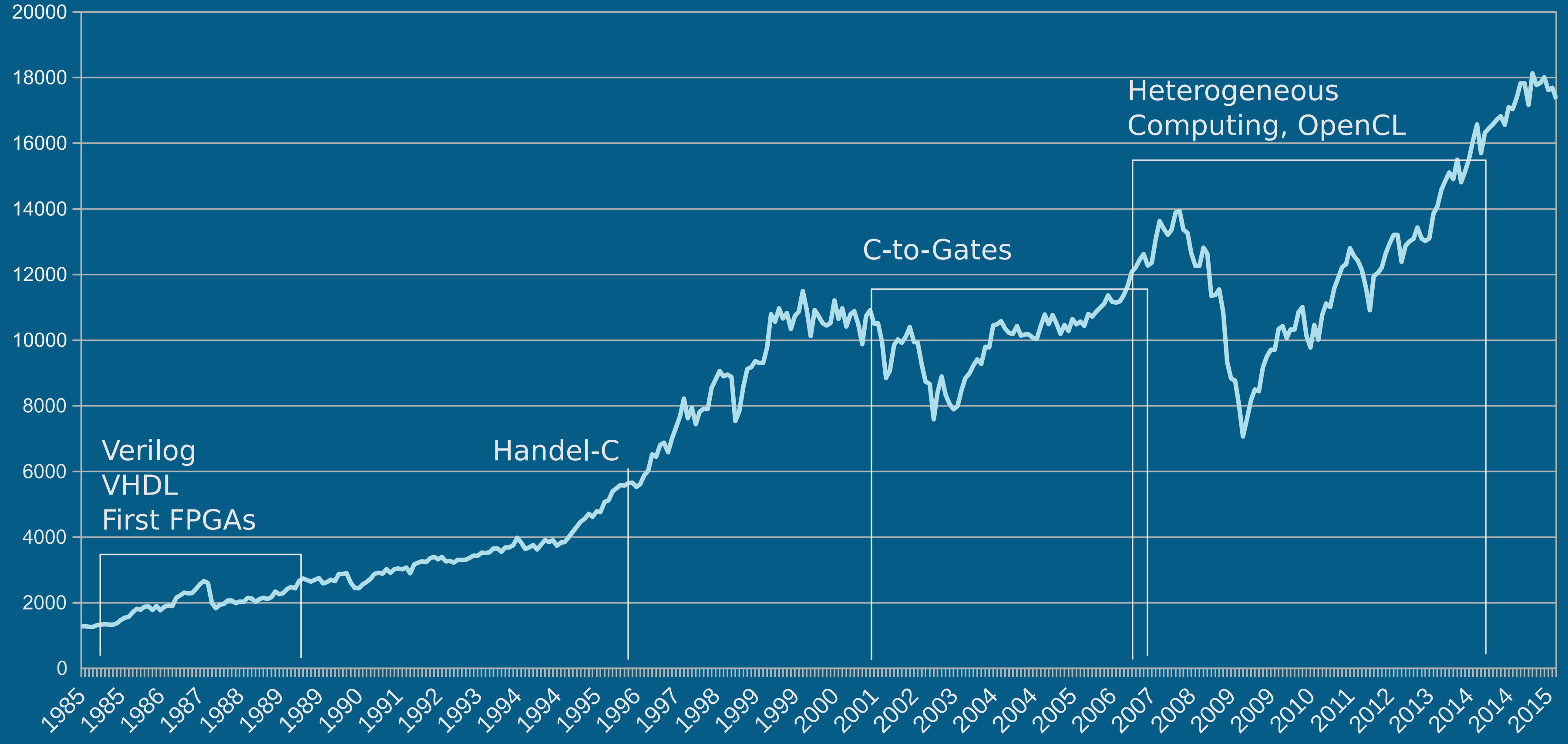
2009: OpenCL ,Apple Inc., Khronos Group

2010: Intel Many Integrated Cores

2011: Altera OpenCL

2015: Xilinx OpenCL

FPGA programming evolution



Dow Jones index, 1985-2015

Where to next?

High-Level Synthesis

- For many years, Verilog/VHDL were good enough
- Then the complexity gap created the need for HLS.
- This reflects the rationale behind VHDL:

  "A language with a wide range of descriptive capability that was independent of technology or design methodology."

- What is lacking in this requirement is "capability for scalable abstraction".

“C to Gates”

- "C-to-Gates" offered that higher abstraction level
- But it was in a way a return to the days before standardised VHDL/Verilog:

> *the various components making up a system were designed and verified using a wide range of different and incompatible languages and tools.*

The Choice of C

- C was designed by Ritchie for the specific purpose of writing the UNIX operating system.
  - i.e. to create a control system for a RAM-based single-threaded system.
  - It is basically a syntactic layer over assembly language.
  - Very different semantics from HDLs
- But it became the *lingua franca* for engineers, and hence the de-facto language for HLS tools.

Really C?

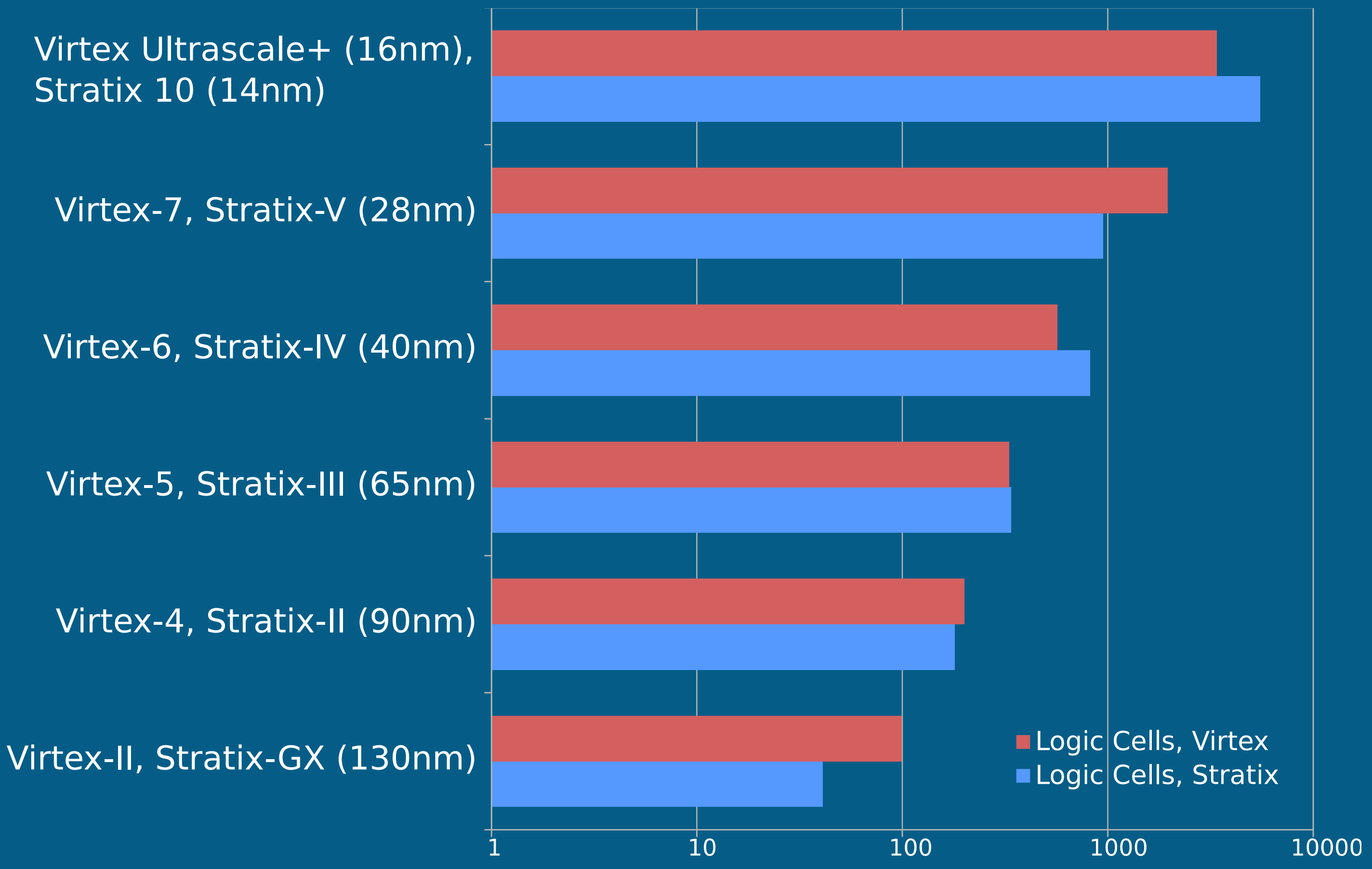
- None of them was ever really C though:
  - "C/C++ with restrictions and pragmas” (e.g. DIME-C, Vivado)
  - “C with restrictions and a CSP API” (e.g. Impulse-C)
  - "C-syntax language with parallel and CSP semantics” (e.g. Handel-C, MaxJ)
- Typically, no recursion, function pointers (no stack) and dynamic allocation (no OS)
  - Until George came along …

Heterogeneous Computing

GPUs, Manycores and FPGAs

- Accelerators attached to host systems have become increasingly popular
- Mainly GPUs,
- But increasingly manycores (MIC, Tilera)
- And FPGAs

FPGA Size Evolution Since 2003



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

Wim Vanderbauwhede

School of Computing Science, University of Glasgow, UK

http://www.slideshare.net/WimVanderbauwhede

Heterogeneous Programming

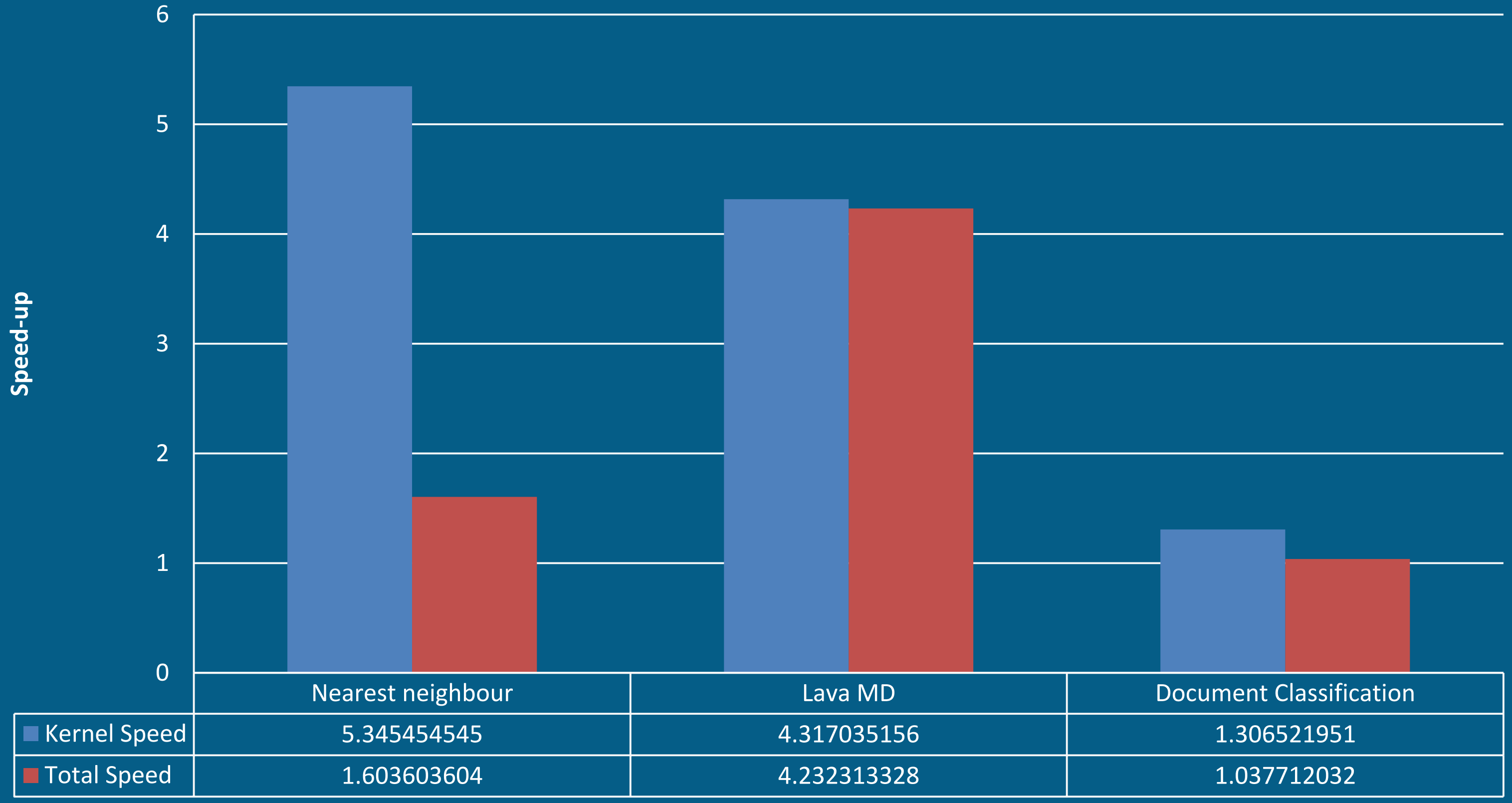
- State of affairs today:
    - Programmer must decide what to offload
    - Write host-accelerator control and data movement code using dedicated API
    - Write accelerator code using dedicated language
    - Many approaches (CUDA, OpenCL, MaxJ, C++ AMP)

Programming Model

- All solutions assume data parallelism:
  - Each kernel is single-threaded, works on a portion of the data
  - Programmer must identify these portions and the amount of parallelism
- So not ideal for FPGAs
- Recent OpenCL specifications have kernel pipes allowing construction of pipelines
- Also support for unified memory space

Performance

OpenCL-FPGA Speed-up vs OpenCL-CPU

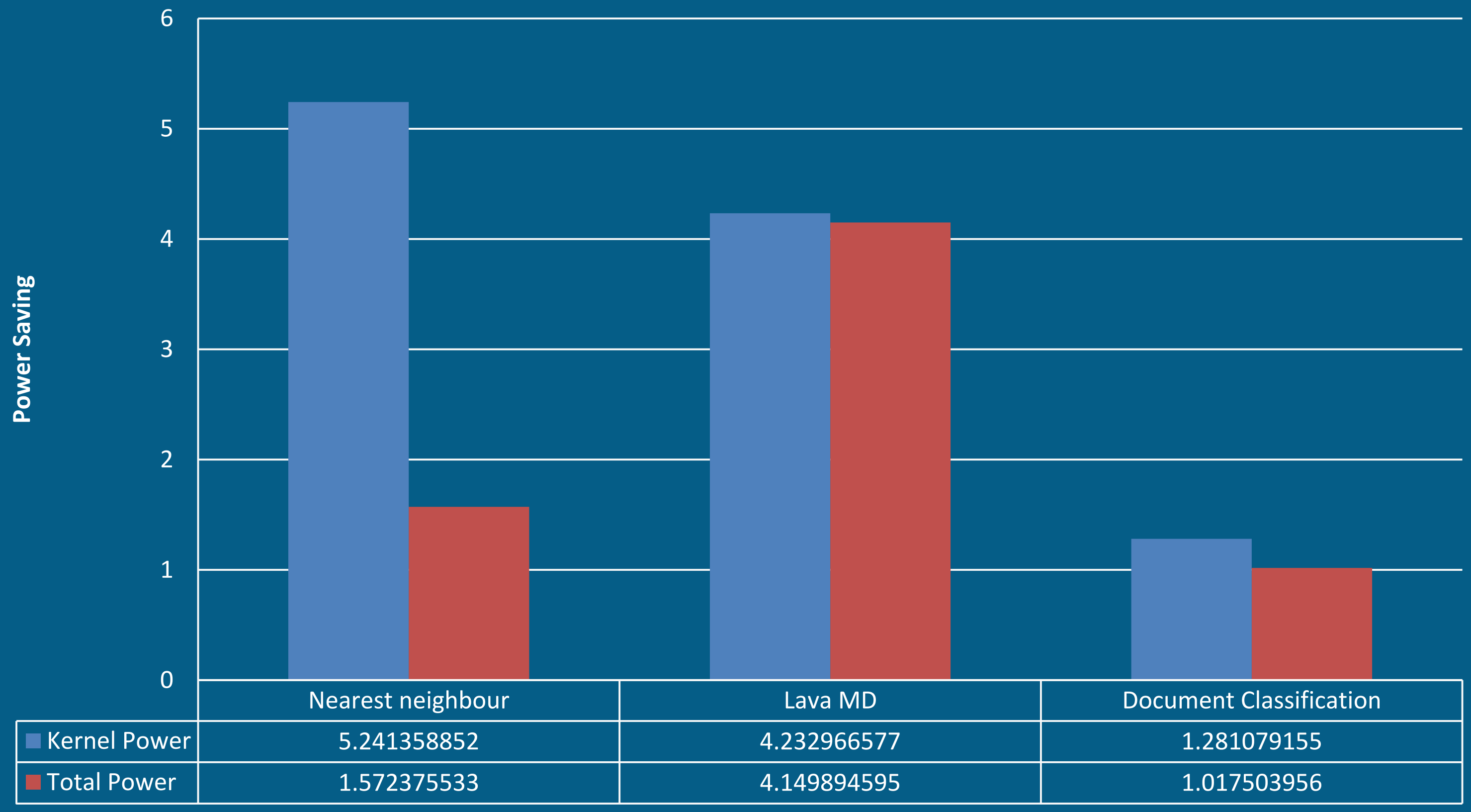


	Nearest neighbour	Lava MD	Document Classification
🟦 Kernel Speed	5.345454545	4.317035156	1.306521951
🟥 Total Speed	1.603603604	4.232313328	1.037712032

Segal, Oren, Nasibeh Nasiri, Martin Margala, and Wim Vanderbauwhede. "High level programming of FPGAs for HPC and data centric applications." *Proc. IEEE HPEC 2014*, pp. 1-3.

Power Savings

CPU/FPGA Power Consumption Ratio



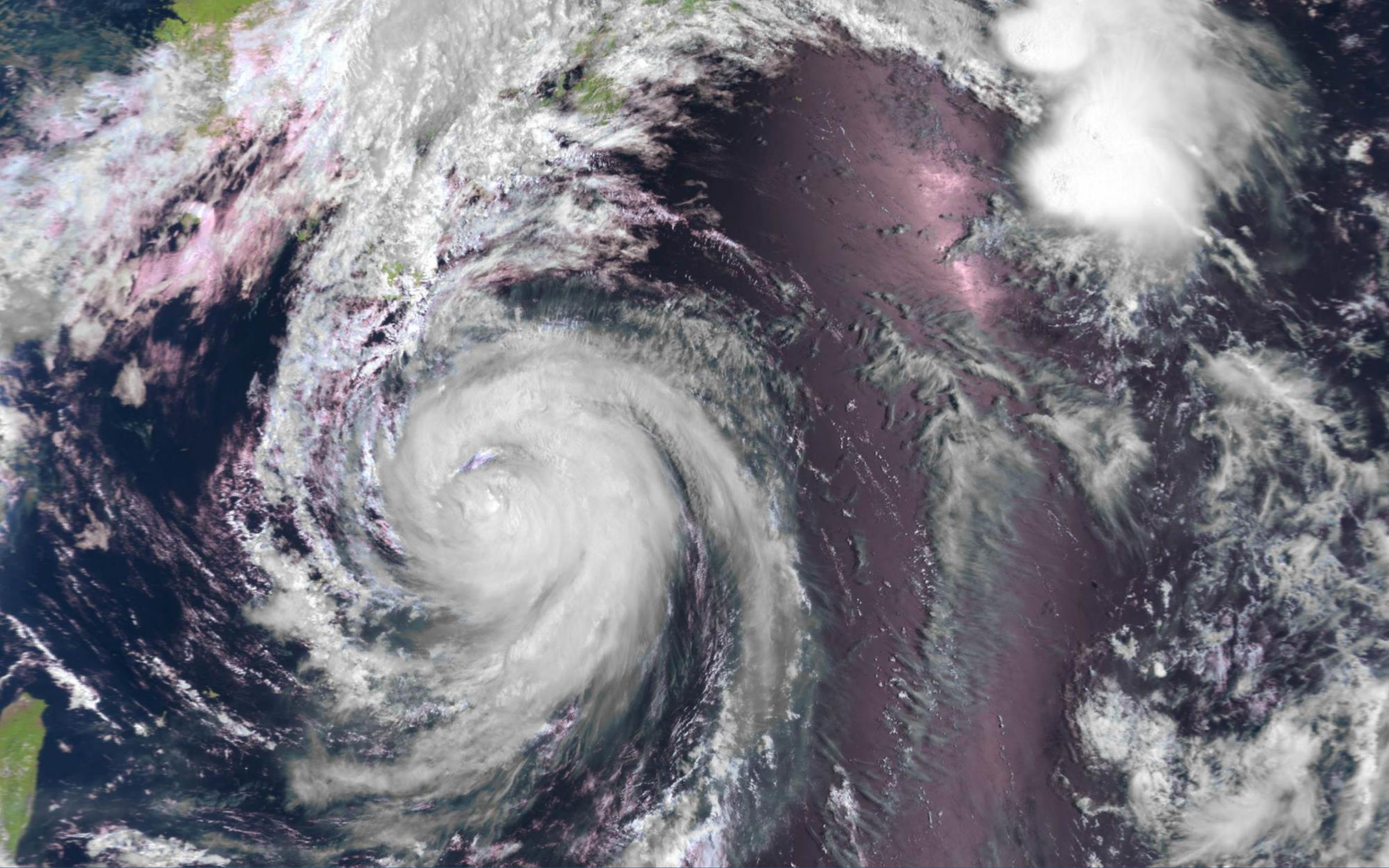
	Nearest neighbour	Lava MD	Document Classification
🟦 Kernel Power	5.241358852	4.232966577	1.281079155
🟥 Total Power	1.572375533	4.149894595	1.017503956

Segal, Oren, Nasibeh Nasiri, Martin Margala, and Wim Vanderbauwhede. "High level programming of FPGAs for HPC and data centric applications." *Proc. IEEE HPEC 2014*, pp. 1-3.

FPGAs as Components in Heterogeneous Systems

Heterogeneous HPC Systems

- Modern HPC cluster node:
  - Multicore/manycore host
  - Accelerators: GPGPU, MIC and increasingly, FPGAs
- HPC workloads
  - Very complex codebase
  - Legacy code



Example: WRF

- Weather Research and Forecasting Model
- Fortran-90, support for MPI and OpenMP
- 1,263,320 lines of code
    - So about ten thousand pages of code listings
- Parts of it have been accelerated manually on GPU (a few thousands of lines)
- Changing the code for a GPU/FPGA system would be a huge task, and the result would not be portable.

FPGAs in HPC

- FPGAs are good at some tasks, e.g.:
  - Bit level, integer and string operations
  - Pipeline parallelism rather than data parallelism
  - Superior internal memory bandwidth
  - Streaming dataflow computations
- But not so good at others
  - Double-precision floating point computations
  - Random memory access computations

Raising the Abstraction Level

One Codebase, Many Components

- For complex HPC applications, FPGAs will never be optimal for the whole codebase
- But neither will multicores or GPUs
- So we need to be able to split the codebase automatically over the different components in the heterogeneous system.
- Therefore, we need to raise the abstraction level beyond “heterogeneous programming” and “high-level synthesis”

Today’s high-level language is tomorrow’s compiler target

- Device-specific high-level abstraction is no longer good enough
- OpenCL is relatively high-level and device-independent, but it is still not good enough
- High-level synthesis languages and heterogeneous programming frameworks should be compilation targets!
- Just like assembly /IR languages and HDLs

Automatic Program Transformations and Cost models

- Starting from a complete, unoptimised program
- Compiler-based program transformations
  - Correct-by-construction
- Component-based, hierarchical cost model for the full system
- Optimization problem:

  find the optimal program variant given the system cost model

A Functional-Programming Approach

- For the particular case of scientific HPC codes
- Focus on array computations
- Express the program using higher-order functions
- Type Transformation based program transformation

Functional Programming

- There are only functions
- Functions can operate on functions
- Functions can return functions
- Syntactic sugar over the λ–calculus

Types in Functional Programming

- Types are just labels to help us reason about the values in a computation
- More general than types in e.g C
- For our purpose, we focus on types of functions that perform array operations
- Functions are values, so they need a type

Examples of Types

-- a function *f* taking a vector of *n* values of type *a* and returing a vector of *m* values of type *b*

```
f : Vec a n -> Vec b m
```

-- a function *map* taking a *function* from *a* to *b* and a vector of type *a*, and returning a vector of type *b*

```
map : (a -> b) -> Vec a n -> Vec b n
```

Type Transformations

- Transform the *type* of a function into another type
- The function transformation can be derived automatically from the type transformation
- The type transformations are provably correct
- Thus the transformed program is correct by construction!

Array Type Transformations

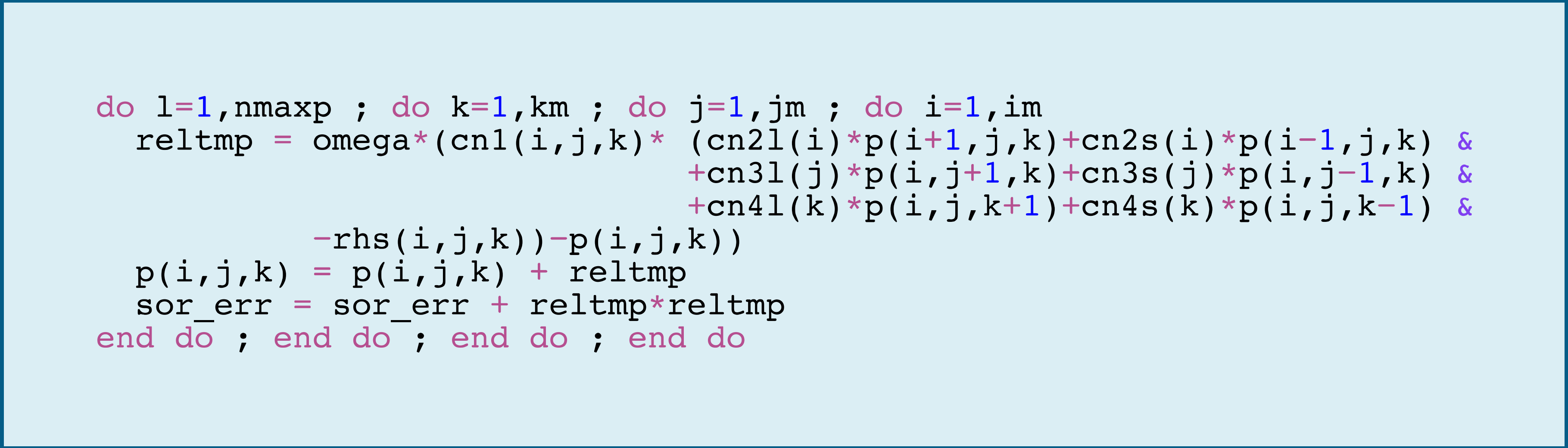
- For this talk, focus on
  - Vector (array) types
  - FPGA cost model
- Programs must be composed using particular higher-order functions (correctness conditions)
- Transformations essentially reshape the arrays

Higher-order Functions

- map: perform a computation on all elements of an array independently, e.g. square all values.
    - can be done sequentially, in parallel or using a pipeline if the computation is pipelined
- foldl: reduce an array to a value using an accumulator, e.g. sum all values.
    - can be done sequentially or, if the computation is associative, using a binary tree

Example: SOR

- Successive Over-Relaxation (SOR) kernel from a Large-Eddy simulator (weather simulation) in Fortran:



```
do l=1,nmaxp ; do k=1,km ; do j=1,jm ; do i=1,im
  reltmp = omega*(cn1(i,j,k)* (cn2l(i)*p(i+1,j,k)+cn2s(i)*p(i-1,j,k) &
                              +cn3l(j)*p(i,j+1,k)+cn3s(j)*p(i,j-1,k) &
                              +cn4l(k)*p(i,j,k+1)+cn4s(k)*p(i,j,k-1) &
           -rhs(i,j,k))-p(i,j,k))
  p(i,j,k) = p(i,j,k) + reltmp
  sor_err = sor_err + reltmp*reltmp
end do ; end do ; end do ; end do
```

Example: SOR using *map*

- Fortran code rewritten as a map of a function over a 1-D vector

```
pps = prepare_vectors p rhs cn1 cn2l ...
ps = map p_sor pps
p_sor pt = reltmp + p_c
  where
     (p_i_p1,...,p_c,rhs_c) = pt
     reltmp = omega * (cn1 * (
                   cn2l_x * p_i_p1 + cn2s_x * p_i_m1
                 + cn3l_x * p_j_p1 + cn3s_x * p_j_m1
                 + cn4l_x * p_k_p1 + cn4s_x * p_k_m1 ) - rhs_c) - p_c
```

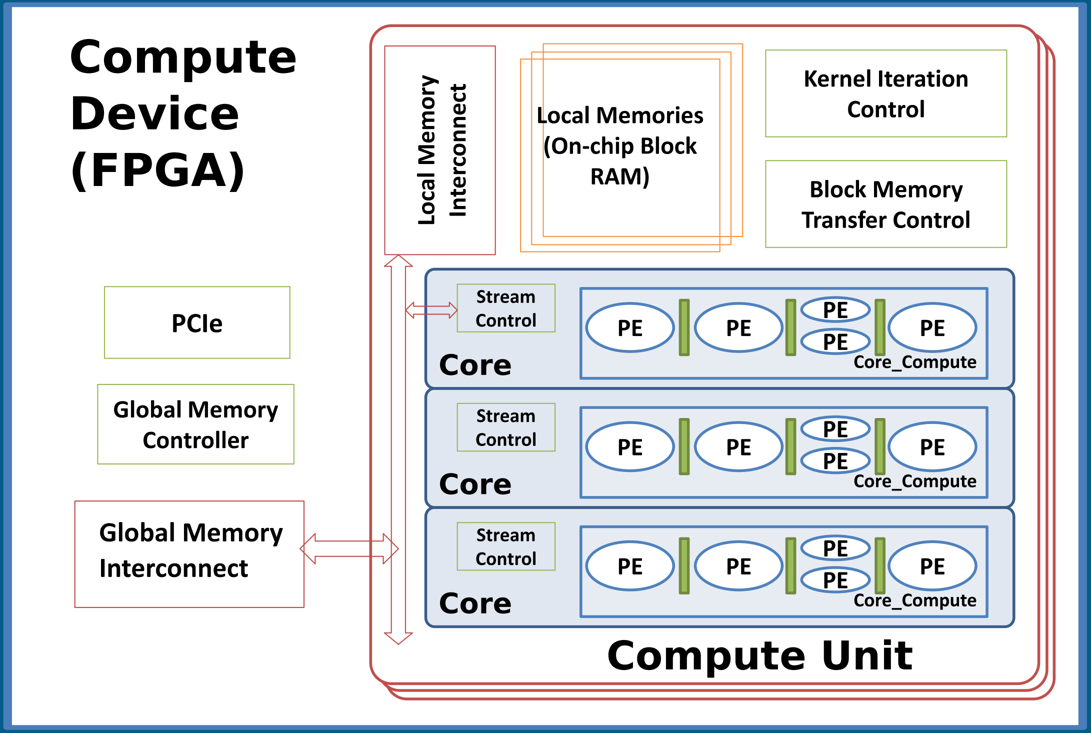
Example: Type Transformation

- Transform the 1-D vector into a 2-D vector
- The program transformation is derived

```
pps  : Vect  (im*jm*km)  t         -- 1D vector
ppst : Vect  km  (Vect  im*jm  t)  -- transformed 2D vector

-- Resulting in a corresponding change in the program:

ps = map  p_sor  pps             -- original program

ppst = reshapeTo  km  pps        -- reshaping data
pst  = map  (map  p_sor)  ppst   -- new program
```

FPGA Cost Modeling

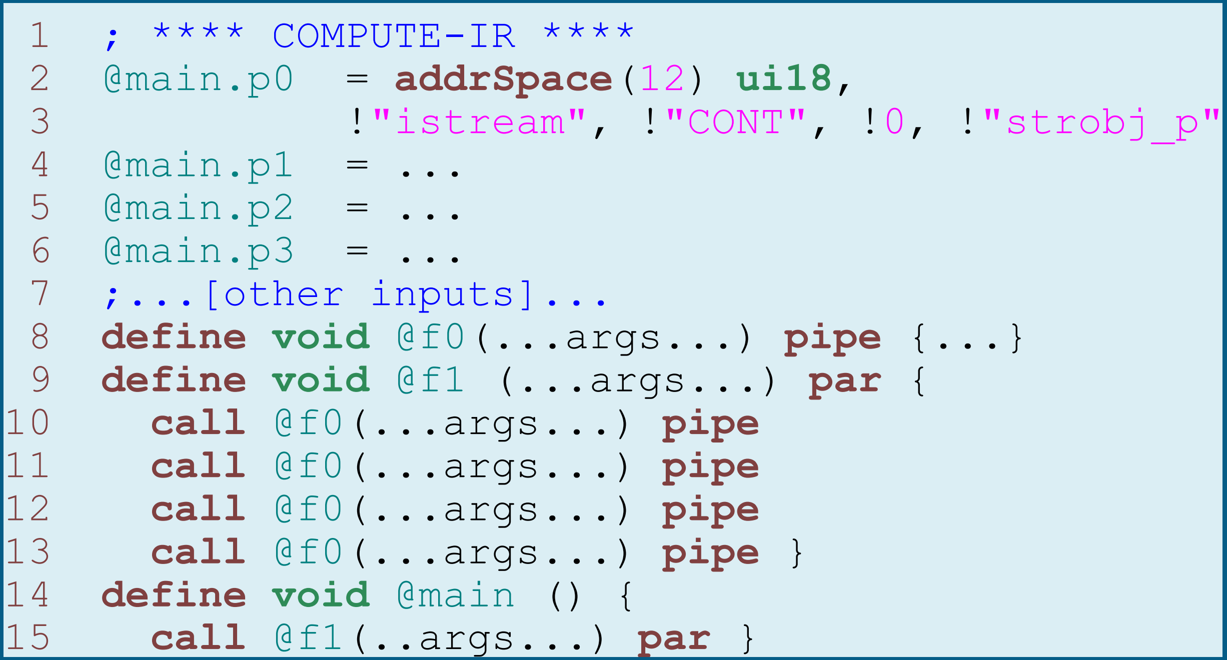
- Based on an overlay architecture 😊



Cost Calculation

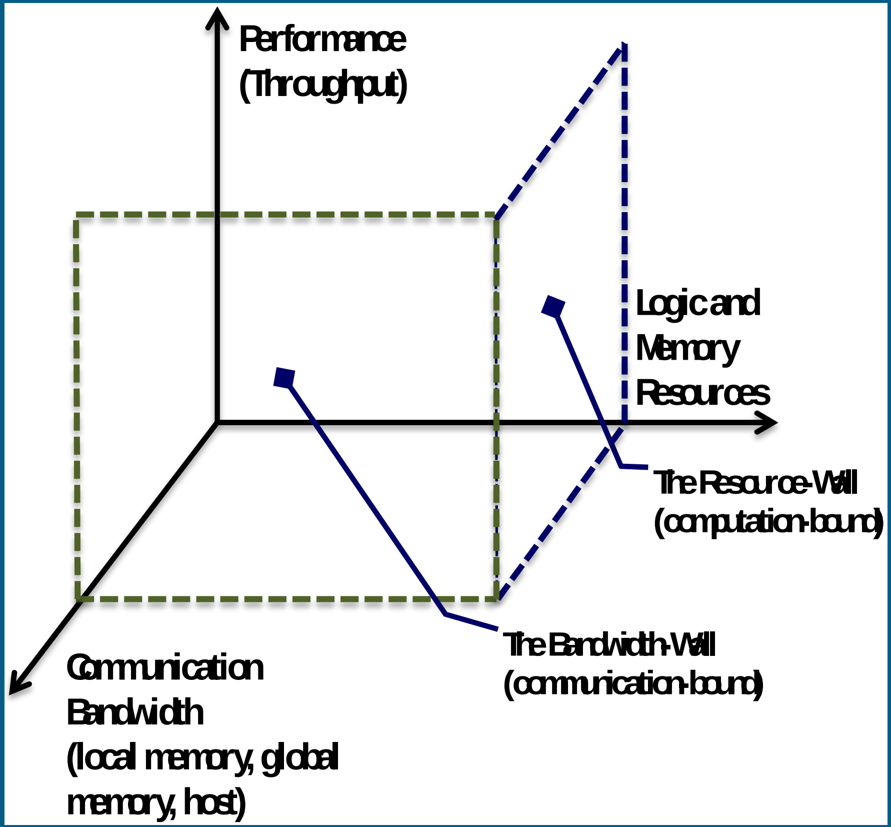
- Uses an Intermediate Representation Language, the TyTra-IR
- TyTra-IR uses LLVM syntax but can express sequential, parallel and pipeline semantics
- Thus a direct mapping to the higher-order functions
- But the cost of computations and communication can be computed directly from the TyTra-IR program
- No need for synthesis

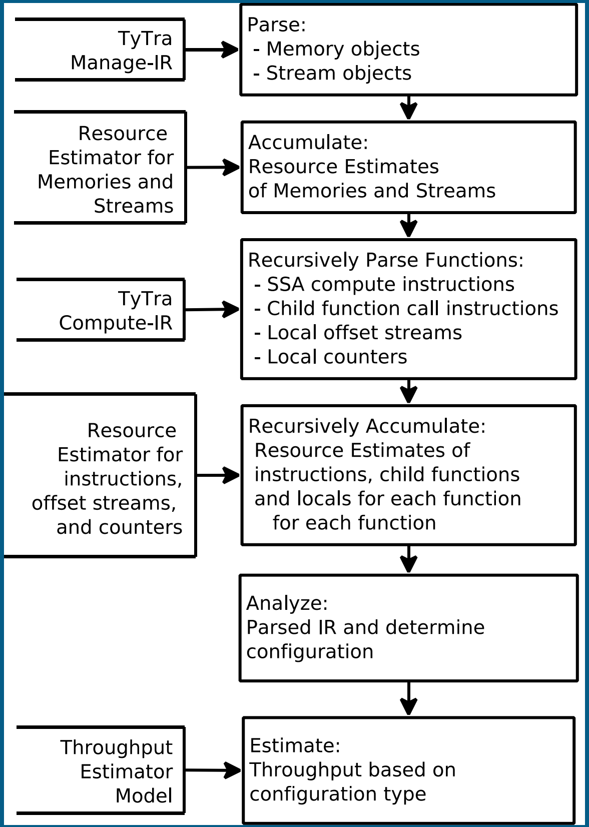
TyTra-IR Example



```
; **** COMPUTE-IR ****
@main.p0  = addrSpace(12) ui18,
          !"istream", !"CONT", !0, !"strobj_p"
@main.p1  = ...
@main.p2  = ...
@main.p3  = ...
;...[other inputs]...
define void @f0(...args...) pipe {...}
define void @f1 (...args...) par {
  call @f0(...args...) pipe
  call @f0(...args...) pipe
  call @f0(...args...) pipe
  call @f0(...args...) pipe }
define void @main () {
  call @f1(..args...) par }
```

Cost Space and Cost Estimation


Performance (Throughput)
Logic and Memory Resources
The Resource-Wall (computation-bound)
The Bandwidth-Wall (communication-bound)
Communication Bandwidth (local memory, global memory, host)



TyTra Manage-IR

Parse:
- Memory objects
- Stream objects

Resource Estimator for Memories and Streams

Accumulate:
Resource Estimates of Memories and Streams

TyTra Compute-IR

Recursively Parse Functions:
- SSA compute instructions
- Child function call instructions
- Local offset streams
- Local counters

Resource Estimator for instructions, offset streams, and counters

Recursively Accumulate:
Resource Estimates of instructions, child functions and locals for each function for each function

Analyze:
Parsed IR and determine configuration

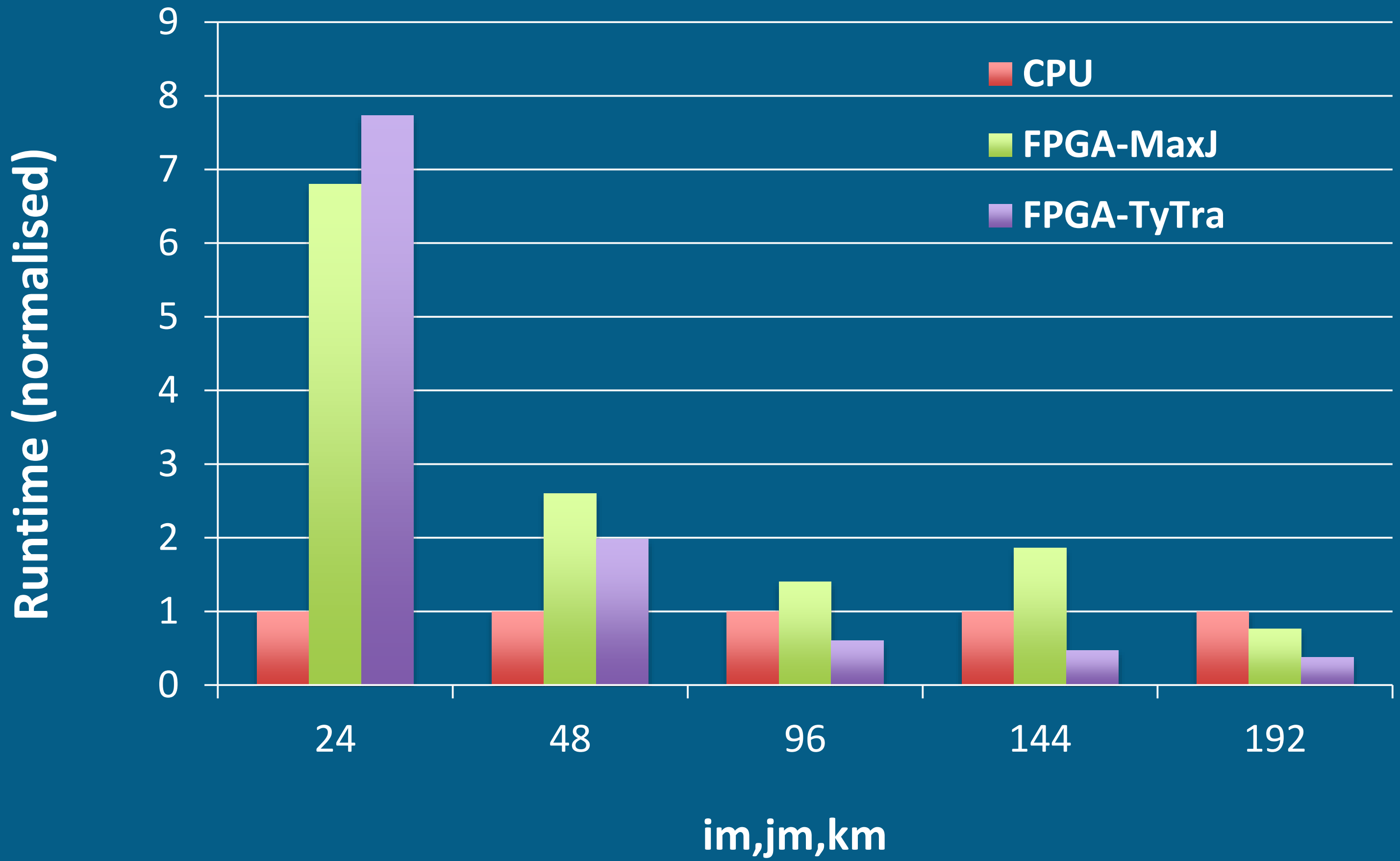
Throughput Estimator Model

Estimate:
Throughput based on configuration type

Cost Model Accuracy

Resource	1-lane(E)	1-lane(G)	4-lane(E)	4-lane(G)
ALUTs	239	164	148K	146K
REGs	725	572	76628	77260
BRAM(bits)	186K	186K	449K	682K
DSPs	9	12	36	24
Cycles/Kernel	1746	1742	436	446
EWGT	190K	222K	763K	488K

Some Results



Full-Program Transformation

- Type Transformations are not FPGA-specific
- Compiler can create variants for full program
- Then separate out parts of the program based on minimal cost on given components of the system
- For parallelisation over a cluster, use Multi-Party Session Types to transform the program into communicating processes

Conclusions

- FPGAs have reached maturity as HPC platforms
- High-Level Synthesis and Heterogeneous Programming are both very important steps forward, and performance is already impressive
- But we need to raise the abstraction level even more
  - Full-system compilers for heterogeneous systems
  - FPGAs are merely components in such systems
  - Type Transformations are one possible way

Thank you!