Introduction to Dataflow Computing

Maxeler Technologies

Code Carpentry Workshop
Peter Sanders, July 2015
Computing on FPGAs
Programmable Spectrum

Control-flow processors

Single-Core CPU  Multi-Core  Several-Cores  Many-Cores  Dataflow processor e.g. FPGA

Increasing Parallelism (#cores)

Increasing Core Complexity

Intel, AMD  GPU (NVIDIA, AMD)  Tilera, XMOS etc...  Maxeler  Hybrid e.g. AMD Fusion, IBM Cell
Control-flow processor (CPU)

Main Memory (RAM)

Execution (ALU)

Instructions

Data

Von Neumann Architecture

Fetch Instruction

Fetch Data

Execute

Store Data

Time
Modern Control-flow processor (CPU)

- Main Memory (RAM)
- Instruction Pipeline
- Execution (ALU)
- Pre-fetch & Branch Prediction
- Instructions
- Data
- Reordering
Where silicon is used?

Intel 6-Core X5680 “Westmere”
Where silicon is used?

Intel 6-Core X5680 “Westmere”

Dataflow Processor
e.g. FPGA

Computation

MaxelerOS

Computation
(Dataflow cores)
On chip resources

• Each application has a different configuration of dataflow cores
• Dataflow cores are built out of basic operating resources on-chip

DSP Resource

General Logic Resource

RAM Resource (20TB/s)
Control flow Microprocessor (CPU)
Dataflow Engine (DFE) - ‘Spatial Computing’
Acceleration Potential

- Ten times slower clock

- Degrees of Freedom
  - Architecture
  - Data type

- Massive parallelism
  - Bit level
  - Pipeline level
  - Architecture level
  - System level
Explaining Control Flow versus Data Flow

Analogy 1: The Ford Production Line

• Experts are expensive and slow (control flow)
• Many specialized workers are more efficient (data flow)
public class MyKernel extends Kernel {

    public MyKernel (KernelParameters parameters) {
        super(parameters);
        DFEVar x = io.input("x", dfeInt(32));
        DFEVar result = x * x + 30;
        io.output("y", result, dfeInt(32));
    }
}

A Dataflow Kernel
Streaming Data through the Kernel

```
5 4 3 2 1 0
```

```
x
```

```
0
```

```
* 0
```

```
30
```

```
+ 30
```

```
y
```

```
30
```

Streaming Data through the Kernel
Streaming Data through the Kernel

5 4 3 2 1 0

x

2

* 4

+ 30

y

30 31 34
Streaming Data through the Kernel

```
5 4 3 2 1 0
```

```
x
```

```
3
```

```
*
```

```
9
```

```
+
```

```
39
```

```
y
```

```
30 31 34 39
```
Streaming Data through the Kernel

```
5 4 3 2 1 0
```

```
x
```

```
4
```

```
* 16
```

```
+ 46
```

```
y
```

```
30 31 34 39 46
```
Streaming Data through the Kernel

Diagram:

- Variables: x, y
- Operations: *
- Values: 5, 25, 30, 35, 55
- Output: 30, 31, 34, 39, 46, 55
Streaming Data through the Kernel

Total Latency = L
Pipelining Data through the Kernel

Add registers into the Dataflow graph.

On FPGAs the shorter the physical distance between registers the higher the clock frequency can be.

Fanout Latency = L1

Multiply Latency = L2

Add Latency = L3

L1 + L2 + L3 > L
Pipelining Data through the Kernel

5 4 3 2 1 0

x

0

* 30

+ y


Pipelining Data through the Kernel

5 4 3 2 1 0

x

1

*

0

30

+

y

[

]
Pipelining Data through the Kernel

5 4 3 2 1 0

x

2

* 1

30

+

y

30
Pipelining Data through the Kernel

5 4 3 2 1 0

\[ x \]

\[ * \]

\[ + \]

\[ 30 \]

\[ 31 \]

\[ 30 \ 31 \]
Pipelining Data through the Kernel
Pipelining Data through the Kernel

\[ x \]
\[ 5 \]
\[ * \]
\[ 16 \]
\[ + \]
\[ 39 \]
\[ y \]
\[ 30 \]
\[ 30 \ 31 \ 34 \ 39 \]
No registers – low latency

Pipelined – highly parallelised
We need to get oil from our oil well to the refinery.

Demand for our oil is rising – we need a faster truck!

We need to process faster – we need higher clock frequency!

So we get a truck.

We fetch the data in small chunks.

Explaining Control Flow Versus Dataflow
Analogy 2: Trucks versus Pipeline

Processor
Oil Refinery

Click to advance to next slide

Memory
Oil Well

We fetch the data in small chunks.
Explaining Control Flow Versus Dataflow
Analogy 2: Trucks versus Pipeline

So we splash out on a Ferrari to carry our oil!
Let's take the time to build a pipeline
Let's build a dataflow computer for this application.
Once we starting pumping, it takes a while to fill up...
The latency of the first result can be high...

Explaining Control Flow Versus Dataflow
Analogy 2: Trucks versus Pipeline

Processor
Oil Refinery

Memory
Oil Well

Click to advance to next slide

Once we starting pumping, it takes a while to fill up...
The latency of the first result can be high...
But then the oil flows constantly.
And we get a result every clock cycle.
Using FPGAs
Traditionally FPGA were for specialists

- Can’t use FPGA on its own ... 
  - Need interface to Computer & Data 
  - Need interface to local memory 
  - Need to design bespoke HW 

- FPGAs difficult to program ... 
  - Specialist languages VHDL, Verilog 
  - Need Electronics training & understand FPGAs 
  - Simulation only at HW level, modelsim 
  - Need also to engineer the interfaces 

- FPGAs difficult to use ... 
  - Need low level drivers to reconfigure and setup 
  - Need efficient and secure interface design
Maxeler solutions

Standard Hardware across FPGA generations and vendors.

Development environment in extended Java language, MaxJ, with fast simulation and debugging tools.

Runtime library and drivers for reconfiguration, monitoring and probing.
Maxeler Data Flow Engines (DFEs)

MAIA

- Stream to/from Host (PCIe)
- Stream to/from DRAM
- Stream to/from DFE (MaxRing)

- 28nm process
- 250MHz clock frequency
- 6.25MB SRAM
- 4,000 multipliers
- 700K logic cells
- 3GB/s CPU bandwidth
- 96GB DRAM

ISCA

- Stream to/from network
  - UDP & TCP

- 28nm process
- 250MHz clock
- Sub 1us latency
- 16K TCP connections
Maxeler Hardware Solutions

**MPC C Series**
- CPUs plus DFEs
- Intel Xeon CPU cores and up to 6 DFEs with 288GB of RAM

**MPC X Series**
- DFEs shared over Infiniband
- Up to 8 DFEs with 384GB of RAM and dynamic allocation of DFEs to CPU servers

**MPC N Series**
- Low latency connectivity
- Intel Xeon CPUs and 1-2 DFEs with up to six 10Gbit Ethernet connections

---

**MaxWorkstation**
- Desktop development system

**MaxCloud**
- On-demand scalable accelerated compute resource, hosted in London
MPC-X2000

- 8 dataflow engines (192-384GB RAM)
- High-speed MaxRing
- Zero-copy RDMA between CPUs and DFEs over Infiniband
- Dynamic CPU/DFE balancing
STFC

Hartree Centre
Software
Accelerating Real Applications

- The majority of lines of code in most applications are unchanged
- CPUs are good for: latency-sensitive, control-intensive, non-repetitive code
- Dataflow engines are good for: high throughput repetitive processing on large data volumes

⇒ A system should contain both

<table>
<thead>
<tr>
<th></th>
<th>Lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Application</td>
<td>1,000,000</td>
</tr>
<tr>
<td>Kernel to accelerate</td>
<td>2,000</td>
</tr>
<tr>
<td>Software to restructure</td>
<td>20,000</td>
</tr>
</tbody>
</table>
Creating custom DFE configurations

Start

- Original Application
  - Identify code for acceleration and analyze bottlenecks
  - Transform app, architect and model performance
  - Write MaxCompiler code
  - Integrate with CPU code
  - Simulate

- Accelerated Application
  - Meets performance goals?
  - Build DFE
  - Functions correctly?

- YES
- NO
MaxCompiler & MaxIDE

• Complete development environment for Maxeler DFE accelerator platforms
• Write *MaxJ* code to describe the dataflow graph
  – *MaxJ* is an extension of Java for MaxCompiler
  – *Execute* the Java to *generate* the DFE image (bitstream)
  – Meta-programming. Java does NOT execute when running final application.
• Compiler generates C API for CPUs to use *the DFE*
  – *C API called SLiC*
  – *Basic Static interface* – *single function*
    • *loads DFE with bitstream*
    • *Sets scalars and streams data in/out*
Application Components

- **Host application (C, Python, Matlab..)**
- **CPU**
  - **SLiC**
  - **MaxelerOS**
- **PCI Express**
- **DFE**
  - **Kernels (MaxJ)** (instantiate the arithmetic structure)
  - **Manager (MaxJ)** (arrange the data orchestration)
- **Memory**
DFE contains a Manager and Kernels

- Globally Asynchronous Locally Synchronous (GALS) architecture.
- Manager has full flow control.
- Manager made from standard blocks.
- Kernels are fully synchronous.
- Kernels runs while data at inputs and space at output, else stalls.
The required parts to create a DFE

CPU Code

Manager Code

Kernel Code
for (int i = 0; i < DATA_SIZE; i++)
y[i] = x[i] * x[i] + 30;

\[ y_i = x_i \times x_i + 30 \]
Development Process

Host Code (.c)

int*x, *y;
for (int i =0; i < DATA_SIZE; i++)
y[i]= x[i] * x[i] + 30;

MyManager (.maxj)

Manager m = new Manager();
Kernel k =
    new MyKernel();
m.setKernel(k);
m.setIO(
    link("x", CPU),
    link("y", CPU));
m.build();

MyKernel (.maxj)

DFEVar x = io.input("x", dfelnt(32));
DFEVar result = x * x + 30;
io.output("y", result, dfelnt(32));
int* x, *y;
MyKernel( DATA_SIZE, 
    x, DATA_SIZE*4);

Manager m = new Manager();
Kernel k = 
    new MyKernel();
m.setKernel(k);
m.setIO( 
    link("x", CPU), 
    link("y", DRAM_LINEAR1D));
m.build();

DFEVar x = io.input("x", dfelnt(32));
DFEVar result = x * x + 30;
io.output("y", result, dfelnt(32));
Data flow graph as generated by compiler 4866 nodes

Each node represents an operator in MaxJ code with area time parameters. Each line (edge) represents a DFEVar in MaxJ code.
Path Latency Reporting

• MaxCompiler gives detailed latency annotation back to the programmer

```javascript
27:  
29:  
31:  
```

12.8ns + 6.4ns = 19.2ns (total compute latency)

• Evaluate precise effect of code on latency
Resource Usage Reporting

• Allows you to see what lines of code are using what resources and focus optimization
  – Separate reports for each kernel and for the manager

<table>
<thead>
<tr>
<th>LUTs</th>
<th>FFs</th>
<th>BRAMs</th>
<th>DSPs</th>
<th>: MyKernel.java</th>
</tr>
</thead>
<tbody>
<tr>
<td>727</td>
<td>871</td>
<td>1.0</td>
<td>2</td>
<td>resources used by this file</td>
</tr>
<tr>
<td>0.24%</td>
<td>0.15%</td>
<td>0.09%</td>
<td>0.10%</td>
<td>% of available</td>
</tr>
<tr>
<td>71.41%</td>
<td>61.82%</td>
<td>100.00%</td>
<td>100.00%</td>
<td>% of total used</td>
</tr>
<tr>
<td>94.29%</td>
<td>97.21%</td>
<td>100.00%</td>
<td>100.00%</td>
<td>% of user resources</td>
</tr>
</tbody>
</table>

: public class MyKernel extends Kernel {
  :   public MyKernel (KernelParameters parameters) {
      :     super(parameters);
      :     DFEVar p = io.input("p", dfeFloat(8,24));
      :     DFEVar q = io.input("q", dfeUInt(8));
      :     DFEVar offset = io.scalarInput("offset", dfeUInt(8));
      :     DFEVar addr = offset + q;
      :     DFEVar v = mem.romMapped("table", addr,
                                      dfeFloat(8,24), 256);
      :     p = p * p;
      :     p = p + v;
      :     io.output("r", p, dfeFloat(8,24));
      :   }
  : }

: public class MyKernel extends Kernel {
  :   public MyKernel (KernelParameters parameters) {
      :     super(parameters);
      :     DFEVar p = io.input("p", dfeFloat(8,24));
      :     DFEVar q = io.input("q", dfeUInt(8));
      :     DFEVar offset = io.scalarInput("offset", dfeUInt(8));
      :     DFEVar addr = offset + q;
      :     DFEVar v = mem.romMapped("table", addr,
                                      dfeFloat(8,24), 256);
      :     p = p * p;
      :     p = p + v;
      :     io.output("r", p, dfeFloat(8,24));
      :   }
  : }
Example Projects
Imaging Platform Example: Weather

![Surface pressure maps from GRADS 2.0.1]  

1U CPU Node  
Wall Clock Time: 2 hours

1U Dataflow Node  
less than 2 minutes

Problem size: (Longitude) 13,600 Km x (Latitude) 3330 Km  
Simulation of baroclinic instability after 500 time steps.

Achieved Computational Speedup for the entire application (not just the kernel) compared to Intel server

**RTM with Chevron**
VTI 19x and TTI 25x

**Sparse Matrix**
20-40x

**Seismic Trace Processing**
24x

**Credit**
J.P. Morgan
32x and Rates 26x

**Lattice Boltzeman**
Fluid Flow 30x

**Conjugate Gradient Opt**
26x
Maxeler UP

- University program has over 150 university members
- Membership is free
- Hardware can be bought with discount; software free
  - Low cost Galava DFE
- Possible to access via simulator and cloud
- Shared research among the members.
Maxeler University Program Members
Maxeler University Program Members

150 Universities on 5 continents.
Number of example applications with, in many cases, access to source and docs.
github.com/maxeler/maxpower

Open source project of kernel utilities and functional blocks.
Maxeler Developer Exchange (MDX)

Google group for Q&A amongst developers and Maxeler staff.

Developer Exchange

The Maxeler Developer Exchange (MDX) is an online forum for the exchange of technical information, questions and answers between developers working with Maxeler acceleration solutions. Membership is open to MaxCompiler developers at Maxeler partners, clients and MAX-UP university program member universities.

You will need a Google account to access MDX.
If you are already logged into your Google account you can access MDX directly here.

If you are not logged in, access MDX here.
Questions?

Break?