FGPU: An SIMT-Architecture for FPGAs

Muhammed Al Kadi, Benedikt Janssen, Michael Huebner

Ruhr University of Bochum

Chair for Embedded Systems for Information Technology

FPGA’16, Monterey, 23 February 2016
Outline

1 Motivation and Background

2 FGPU Architecture
   - Execution Model
   - Platform Model
   - Compute Unit Architecture
   - Global Memory Controller

3 Implementation and Results

4 Future Work and Conclusion
1. Motivation and Background

2. FGPU Architecture
   - Execution Model
   - Platform Model
   - Compute Unit Architecture
   - Global Memory Controller

3. Implementation and Results

4. Future Work and Conclusion
Motivation

What is FGPU?

- An FPGA-GPU for general purpose computing
- A portable, flexible and scalable soft processor
- A multi-core GPU-like architecture
- Its ISA is designed to support execution of OpenCL kernels
- Capable of interfacing many AXI4-compatible data interfaces with an internal L1 cache
- It does not replicate any other architecture
- Implemented completely in VHDL-2002
Motivation

Why FGPU?

- **Standard programming:** OpenCL-compatible, no PRAGMAs necessary, shorter development cycles

- **Scalable task management:** new tasks occupy no extra area on the FPGA

- **Design space exploration:** not possible with hard embedded GPUs

- **Application specific adaptations:** to achieve the best area/power and performance trade-off

- **High efficiency:** compared to other soft architectures
Motivation

Why FGPU?

- **Standard programming**: OpenCL-compatible, no PRAGMAs necessary, shorter development cycles
- **Scalable task management**: new tasks occupy no extra area on the FPGA
- **Design space exploration**: not possible with hard embedded GPUs
- **Application specific adaptations**: to achieve the best area/power and performance trade-off
- **High efficiency**: compared to other soft architectures
Motivation

Why FGPU?

- **Standard programming**: OpenCL-compatible, no PRAGMAs necessary, shorter development cycles
- **Scalable task management**: new tasks occupy no extra area on the FPGA
- **Design space exploration**: not possible with hard embedded GPUs
- **Application specific adaptations**: to achieve the best area/power and performance trade-off
- **High efficiency**: compared to other soft architectures
Motivation

Why FGPU?

- **Standard programming:** OpenCL-compatible, no PRAGMAs necessary, shorter development cycles
- **Scalable task management:** new tasks occupy no extra area on the FPGA
- **Design space exploration:** not possible with hard embedded GPUs
- **Application specific adaptations:** to achieve the best area/power and performance trade-off
- **High efficiency:** compared to other soft architectures
Motivation

Why FGPU?

- **Standard programming:** OpenCL-compatible, no PRAGMAs necessary, shorter development cycles
- **Scalable task management:** new tasks occupy no extra area on the FPGA
- **Design space exploration:** not possible with hard embedded GPUs
- **Application specific adaptations:** to achieve the best area/power and performance trade-off
- **High efficiency:** compared to other soft architectures
Motivation

Why FGPU?

- **Standard programming:** OpenCL-compatible, no PRAGMAs necessary, shorter development cycles
- **Scalable task management:** new tasks occupy no extra area on the FPGA
- **Design space exploration:** not possible with hard embedded GPUs
- **Application specific adaptations:** to achieve the best area/power and performance trade-off
- **High efficiency:** compared to other soft architectures
SIMT (Single-Instruction Multiple-Treads)

is a parallel execution model to program many-core architectures.

- A single instruction can be concurrently executed by multiple threads.
- Thread are scheduled at runtime on the available cores.

GPGPU (General Purpose Computing on Graphical Processing Units)

- An efficient solution for many application e.g. filtering, scientific simulations, matrix operations, sorting, DSP etc.
- Embedded GPUs are easier to program with OpenCL or CUDA than FPGAs with HDLs.
SIMT (Single-Instruction Multiple-Treads) is a parallel execution model to program many-core architectures.

- A single instruction can be concurrently executed by multiple threads.
- Threads are scheduled at runtime on the available cores.

GPGPU (General Purpose Computing on Graphical Processing Units)

- An efficient solution for many applications e.g. filtering, scientific simulations, matrix operations, sorting, DSP etc.
- Embedded GPUs are easier to program with OpenCL or CUDA than FPGAs with HDLs.
Background
SIMT and GPGPU

SIMT (Single-Instruction Multiple-Treads)

is a **parallel** execution model to program **many-core** architectures.

- A single instruction can be concurrently executed by multiple threads.
- Thread are scheduled at runtime on the available cores.

GPGPU (General Purpose Computing on Graphical Processing Units)

- An efficient solution for many application e.g. filtering, scientific simulations, matrix operations, sorting, DSP etc.
- Embedded GPUs are easier to program with OpenCL or CUDA than FPGAs with HDLs.
Outline

1 Motivation and Background

2 FGPU Architecture
   - Execution Model
   - Platform Model
   - Compute Unit Architecture
   - Global Memory Controller

3 Implementation and Results

4 Future Work and Conclusion
Outline

1 Motivation and Background

2 FGPU Architecture
   - Execution Model
   - Platform Model
   - Compute Unit Architecture
   - Global Memory Controller

3 Implementation and Results

4 Future Work and Conclusion
Threads (Work-items) get coordinated in a 1-, 2- or 3D index space.

Work-items get scheduled together in Work-groups (WGs) on idle cores.

WGs are splitted into Wavefronts (WFs).

WF’s work-items share the same program counter.

Example (Array multiplication):

<table>
<thead>
<tr>
<th>1D Index Space</th>
<th>Thread (Work-item)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>i</td>
</tr>
<tr>
<td>N-1</td>
<td></td>
</tr>
</tbody>
</table>

1st Array:

0 1

2nd Array:

0 1

Result Array:

0 1

0 1

Wavefront

Work-group

Array multiplication example:
FGPU Architecture
Execution Model (OpenCL-Compatible)

- Threads (Work-items) get coordinated in a 1-, 2- or 3D *index space*
- Work-items get scheduled together in *Work-groups (WGs)* on idle cores
- WGs are splitted into *Wavefronts (WFs)*
- WF’s work-items share the same program counter

Example (Array multiplication)

```
<table>
<thead>
<tr>
<th>0</th>
<th>N-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>N-1</td>
</tr>
<tr>
<td>N-1</td>
<td></td>
</tr>
</tbody>
</table>

1D Index Space
```

```
<table>
<thead>
<tr>
<th>0</th>
<th>N-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>N-1</td>
</tr>
</tbody>
</table>

1st Array
```

```
<table>
<thead>
<tr>
<th>0</th>
<th>N-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>N-1</td>
</tr>
</tbody>
</table>

2nd Array
```

```
<table>
<thead>
<tr>
<th>0</th>
<th>N-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>N-1</td>
</tr>
</tbody>
</table>

Result Array
```

```
<table>
<thead>
<tr>
<th>0</th>
<th>N-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>N-1</td>
</tr>
</tbody>
</table>

Thread (Work-item)
```

```
<table>
<thead>
<tr>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
</tr>
<tr>
<td>N-1</td>
</tr>
</tbody>
</table>

Work-group
```

```
<table>
<thead>
<tr>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
</tr>
<tr>
<td>N-1</td>
</tr>
</tbody>
</table>

Wavefront
```
Threads (Work-items) get coordinated in a 1-, 2- or 3D index space

Work-items get scheduled together in Work-groups (WGs) on idle cores

WGs are split into Wavefronts (WFs)

WF’s work-items share the same program counter

Example (Array multiplication)

1D Index Space

1st Array

2nd Array

Result Array

Thread (Work-item)

0 1

0 1

0 1

0 1

0 1

0 1
FGPU Architecture

Execution Model (OpenCL-Compatible)

- Threads (Work-items) get coordinated in a 1-, 2- or 3D index space
- Work-items get scheduled together in Work-groups (WGs) on idle cores
- WGs are splitted into Wavefronts (WFs)
- WF’s work-items share the same program counter

Example (Array multiplication)

```
<table>
<thead>
<tr>
<th>1st Array</th>
<th>2nd Array</th>
<th>Result Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1</td>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>N-1</td>
<td>N-1</td>
<td>N-1</td>
</tr>
</tbody>
</table>
```

Thread (Work-item)
FGPU Architecture

Execution Model (OpenCL-Compatible)

- Threads (Work-items) get coordinated in a 1-, 2- or 3D index space
- Work-items get scheduled together in Work-groups (WGs) on idle cores
- WGs are splitted into Wavefronts (WFs)
- WF’s work-items share the same program counter

Example (Array multiplication)

<table>
<thead>
<tr>
<th>1D Index Space</th>
<th>Thread (Work-item)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>i</td>
</tr>
<tr>
<td>N-1</td>
<td></td>
</tr>
<tr>
<td>1st Array</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2nd Array</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Result Array</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Outline

1. Motivation and Background

2. FGPU Architecture
   - Execution Model
   - Platform Model
     - Compute Unit Architecture
     - Global Memory Controller

3. Implementation and Results

4. Future Work and Conclusion
FGPU accommodates several *Compute Units (CUs)*, each holds a single array of *Processing Elements (PEs)*

- All PEs in a CU execute the same instruction at the same time
- The binary code is executed from the *Code RAM (CRAM)*
- Information that do not belong to the binary, e.g. # of work-items to launch or parameter values, has to be stored in the *Link RAM (LRAM)*
FGPU accommodates several *Compute Units (CUs)*, each holds a single array of *Processing Elements (PEs)*.

- All PEs in a CU execute the same instruction at the same time.
- The binary code is executed from the *Code RAM (CRAM)*.
- Information that do not belong to the binary, e.g. # of work-items to launch or parameter values, has to be stored in the *Link RAM (LRAM)*.
FGPU accommodates several *Compute Units (CUs)*, each holds a single array of *Processing Elements (PEs)*

- All PEs in a CU execute the same instruction at the same time
  - The binary code is executed from the *Code RAM (CRAM)*
  - Information that do not belong to the binary, e.g. # of work-items to launch or parameter values, has to be stored in the *Link RAM (LRAM)*
FGPU accommodates several *Compute Units (CUs)*, each holds a single array of *Processing Elements (PEs)*

All PEs in a CU execute the same instruction at the same time

The binary code is executed from the *Code RAM (CRAM)*

Information that do not belong to the binary, e.g. # of work-items to launch or parameter values, has to be stored in the *Link RAM (LRAM)*
FGPU accommodates several Compute Units (CUs), each holds a single array of Processing Elements (PEs)

- All PEs in a CU execute the same instruction at the same time
- The binary code is executed from the Code RAM (CRAM)
- Information that do not belong to the binary, e.g. # of work-items to launch or parameter values, has to be stored in the Link RAM (LRAM)
- The **WG Dispatcher** assigns WGs on idle CUs
- The **WF Scheduler** admits WFs for execution, e.g. after a memory access is performed
- The **Runtime Memory (RTM)** implements the **Work-Item Built-In Functions** defined by OpenCL, e.g. feeding coordinates in the index space.
The **WG Dispatcher** assigns WGs on idle CUs

- The **WF Scheduler** admits WFs for execution, e.g. after a memory access is performed

- The **Runtime Memory (RTM)** implements the **Work-Item Built-In Functions** defined by OpenCL, e.g. feeding coordinates in the index space.
The *WG Dispatcher* assigns WGs on idle CUs

The *WF Scheduler* admits WFs for execution, e.g. after a memory access is performed.

The *Runtime Memory (RTM)* implements the *Work-Item Built-In Functions* defined by OpenCL, e.g. feeding coordinates in the index space.
The *WG Dispatcher* assigns WGs on idle CUs

The *WF Scheduler* admits WFs for execution, e.g. after a memory access is performed

The *Runtime Memory (RTM)* implements the *Work-Item Built-In Functions* defined by OpenCL, e.g. feeding coordinates in the index space.
The **WG Dispatcher** assigns WGs on idle CUs

The **WF Scheduler** admits WFs for execution, e.g. after a memory access if performed

The **Runtime Memory (RTM)** implements the **Work-Item Built-In Functions** defined by OpenCL, e.g. feeding coordinates in the index space.
Outline

1 Motivation and Background

2 FGPU Architecture
   - Execution Model
   - Platform Model
   - Compute Unit Architecture
   - Global Memory Controller

3 Implementation and Results

4 Future Work and Conclusion
8 WFs can be managed within a single CU

The same instruction gets executed over 8 clock cycles on 8 PEs

A work-item owns 32 registers (32bit)

Register files are held in dual-port RAMs and switched with no latency

The pipeline consists of 18 stages!

Doubled clock frequency for the register files and the ALUs

64 outstanding memory requests can be managed at the same time
8 WFs can be managed within a single CU

The same instruction gets executed over 8 clock cycles on 8 PEs

A work-item owns 32 registers (32bit)

Register files are held in dual-port RAMs and switched with no latency

The pipeline consists of 18 stages!

Doubled clock frequency for the register files and the ALUs

64 outstanding memory requests can be managed at the same time
- 8 WFIs can be managed within a single CU
- The same instruction gets executed over 8 clock cycles on 8 PEs
- A work-item owns 32 registers (32bit)
- Register files are held in dual-port RAMs and switched with no latency
- The pipeline consists of 18 stages!
- Doubled clock frequency for the register files and the ALUs
- 64 outstanding memory requests can be managed at the same time
8 WFs can be managed within a single CU
- The same instruction gets executed over 8 clock cycles on 8 PEs
- A work-item owns 32 registers (32bit)
- Register files are held in dual-port RAMs and switched with no latency
- The pipeline consists of 18 stages!
- Doubled clock frequency for the register files and the ALUs
- 64 outstanding memory requests can be managed at the same time
- 8 WFs can be managed within a single CU
- The same instruction gets executed over 8 clock cycles on 8 PEs
- A work-item owns 32 registers (32bit)
- Register files are held in dual-port RAMs and switched with no latency
- The pipeline consists of 18 stages!
- Doubled clock frequency for the register files and the ALUs
- 64 outstanding memory requests can be managed at the same time
FGPU Architecture

- 8 WFs can be managed within a single CU
- The same instruction gets executed over 8 clock cycles on 8 PEs
- A work-item owns 32 registers (32bit)
- Register files are held in dual-port RAMs and switched with no latency
- The pipeline consists of 18 stages!
- Doubled clock frequency for the register files and the ALUs
- 64 outstanding memory requests can be managed at the same time
- 8 WFs can be managed within a single CU
- The same instruction gets executed over 8 clock cycles on 8 PEs
- A work-item owns 32 registers (32bit)
- Register files are held in dual-port RAMs and switched with no latency
- The pipeline consists of 18 stages!
- Doubled clock frequency for the register files and the ALUs
- 64 outstanding memory requests can be managed at the same time
- 8 WFs can be managed within a single CU
- The same instruction gets executed over 8 clock cycles on 8 PEs
- A work-item owns 32 registers (32bit)
- Register files are held in dual-port RAMs and switched with no latency
- The pipeline consists of 18 stages!
- Doubled clock frequency for the register files and the ALUs
- 64 outstanding memory requests can be managed at the same time
Outline

1. Motivation and Background
2. FGPU Architecture
   - Execution Model
   - Platform Model
   - Compute Unit Architecture
   - Global Memory Controller
3. Implementation and Results
4. Future Work and Conclusion
- It processes up to 64 outstanding requests from multiple CUs
- Not served requests get increasing priorities as they are waiting
- A multi-bank, directed mapped cache with write-back strategy is included
- Accesses to global memory are parallelized over many AXI4-ports
- Data is transferred in bursts that fill a single cache line
It processes up to 64 outstanding requests from multiple CUs

- Not served requests get increasing priorities as they are waiting
- A multi-bank, directed mapped cache with write-back strategy is included
- Accesses to global memory are parallelized over many AXI4-ports
- Data is transferred in bursts that fill a single cache line
It processes up to 64 outstanding requests from multiple CUs

- Not served requests get increasing priorities as they are waiting
- A multi-bank, directed mapped cache with write-back strategy is included
- Accesses to global memory are parallelized over many AXI4-ports
- Data is transferred in bursts that fill a single cache line
It processes up to 64 outstanding requests from multiple CUs

- Not served requests get increasing priorities as they are waiting
- A multi-bank, directed mapped cache with write-back strategy is included
- Accesses to global memory are parallelized over many AXI4-ports
- Data is transferred in bursts that fill a single cache line
FGPU Architecture

Global Memory Controller Architecture

- It processes up to 64 outstanding requests from multiple CUs
- Not served requests get increasing priorities as they are waiting
- A multi-bank, directed mapped cache with write-back strategy is included
- Accesses to global memory are parallelized over many AXI4-ports
- Data is transferred in bursts that fill a single cache line
- It processes up to 64 outstanding requests from multiple CUs
- Not served requests get increasing priorities as they are waiting
- A multi-bank, directed mapped cache with write-back strategy is included
- Accesses to global memory are parallelized over many AXI4-ports
- Data is transferred in bursts that fill a single cache line
Outline

1. Motivation and Background

2. FGPU Architecture
   - Execution Model
   - Platform Model
   - Compute Unit Architecture
   - Global Memory Controller

3. Implementation and Results

4. Future Work and Conclusion
Results

System Setup

- Development board: ZC706 (Zynq XC7Z045) from Xilinx
- Two other solutions for comparison:
  - The hard ARM-Cortex A9
  - The soft MicroBlaze
- Processing data located in the DDR module
- The benchmark was compiled with -O3 as bare-metal application
- Only 32bit integer operations have been considered
- Cache flushing was always performed
Results
System Setup

- Development board: ZC706 (Zynq XC7Z045) from Xilinx
- Two other solutions for comparison:
  - The hard ARM-Cortex A9
  - The soft MicroBlaze
- Processing data located in the DDR module
- The benchmark was compiled with -O3 as bare-metal application
- Only 32bit integer operations have been considered
- Cache flushing was always performed
Results

System Setup

- Development board: ZC706 (Zynq XC7Z045) from Xilinx
- Two other solutions for comparison:
  - The hard ARM-Cortex A9
  - The soft MicroBlaze
- Processing data located in the DDR module
- The benchmark was compiled with -O3 as bare-metal application
- Only 32bit integer operations have been considered
- Cache flushing was always performed
Results

System Setup

- Development board: ZC706 (Zynq XC7Z045) from Xilinx
- Two other solutions for comparison:
  - The hard ARM-Cortex A9
  - The soft MicroBlaze
- Processing data located in the DDR module
- The benchmark was compiled with -O3 as bare-metal application
- Only 32-bit integer operations have been considered
- Cache flushing was always performed
Results
System Setup

- Development board: ZC706 (Zynq XC7Z045) from Xilinx
- Two other solutions for comparison:
  - The hard ARM-Cortex A9
  - The soft MicroBlaze
- Processing data located in the DDR module
- The benchmark was compiled with -O3 as bare-metal application
- Only 32bit integer operations have been considered
- Cache flushing was always performed
Results

System Setup

- Development board: ZC706 (Zynq XC7Z045) from Xilinx
- Two other solutions for comparison:
  - The hard ARM-Cortex A9
  - The soft MicroBlaze
- Processing data located in the DDR module
- The benchmark was compiled with -O3 as bare-metal application
- Only 32bit integer operations have been considered
- Cache flushing was always performed
Results
System Setup

- Development board: ZC706 (Zynq XC7Z045) from Xilinx
- Two other solutions for comparison:
  - The hard ARM-Cortex A9
  - The soft MicroBlaze
- Processing data located in the DDR module
- The benchmark was compiled with -O3 as bare-metal application
- Only 32bit integer operations have been considered
- Cache flushing was always performed
FGPU Implementation

**Highlights**

- **Scalability**
  - Up to 4K threads run simultaneously on 64 PEs
  - Slight degradation for the operating frequencies for bigger designs

- **Portability**
  - No IP-cores or primitives
  - Even DSP slices in pipeline mode were targeted without using any IP-cores

- **Flexibility**
  - Many parameters can be configured to meet the best performance/area trade-off

.Floorplan for 8 CUs on XC7Z045.
FGPU Implementation

Highlights

**Scalability**
- Up to 4K threads run simultaneously on 64 PEs
- Slight degradation for the operating frequencies for bigger designs

**Portability**: no IP-cores or primitives
- Even DSP slices in pipeline mode were targeted without using any IP-cores

**Flexibility**
- Many parameters can be configured to meet the best performance/area trade-off

Floorplan for 8 CUs on XC7Z045
FGPU Implementation

Highlights

- **Scalability**
  - Up to 4K threads run simultaneously on 64 PEs
  - Slight degradation for the operating frequencies for bigger designs

- **Portability**: no IP-cores or primitives
  - Even DSP slices in pipeline mode were targeted without using any IP-cores

- **Flexibility**
  - Many parameters can be configured to meet the best performance/area trade-off

Floorplan for 8 CUs on XC7Z045
FGPU Implementation

**Highlights**

- **Scalability**
  - Up to 4K threads run simultaneously on 64 PEs
  - Slight degradation for the operating frequencies for bigger designs

- **Portability**: no IP-cores or primitives
  - Even DSP slices in pipeline mode were targeted without using any IP-cores

- **Flexibility**
  - Many parameters can be configured to meet the best performance/area trade-off

Floorplan for 8 CUs on XC7Z045
Results

Area Requirements

- Many FGPUs with different settings were tested
- Clock frequencies were fixed at 200MHz/400MHz
- BRAMs and DSPs were targeted but without the manufacturer’s IP-cores
- More FFs than LUTs were consumed due to the deep pipeline
  - Empty pipeline stages were inserted between the two clock domains to improve timing

Adjustable parameters with direct influence on design scalability

<table>
<thead>
<tr>
<th>Module</th>
<th>Parameter</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>CU</td>
<td># CUs</td>
<td>2, 4, 8</td>
</tr>
<tr>
<td></td>
<td># Outstanding mem. requests</td>
<td>16, 24, 32</td>
</tr>
<tr>
<td>Memory Controller</td>
<td>Cache Size</td>
<td>1 to 8KB</td>
</tr>
<tr>
<td></td>
<td># Cache Read Banks</td>
<td>2, 4, 8</td>
</tr>
<tr>
<td></td>
<td># Outstanding mem. requests</td>
<td>32, 64</td>
</tr>
<tr>
<td></td>
<td># AXI4 interfaces</td>
<td>1, 2, 4</td>
</tr>
<tr>
<td></td>
<td># Tag Managers</td>
<td>2, 4, 8, 16</td>
</tr>
</tbody>
</table>

Area requirements for different configurations

<table>
<thead>
<tr>
<th></th>
<th>LUTs</th>
<th>FFs</th>
<th>BRAM</th>
<th>DSPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available</td>
<td>219K</td>
<td>437K</td>
<td>545</td>
<td>900</td>
</tr>
<tr>
<td>8 CUs</td>
<td>57%</td>
<td>36%</td>
<td>31%</td>
<td>21%</td>
</tr>
<tr>
<td>4 CUs</td>
<td>35%</td>
<td>20%</td>
<td>18%</td>
<td>11%</td>
</tr>
<tr>
<td>2 CUs</td>
<td>26%</td>
<td>14%</td>
<td>12%</td>
<td>5.3%</td>
</tr>
<tr>
<td>2 CUs (min)</td>
<td>9.6%</td>
<td>8.1%</td>
<td>8.5%</td>
<td>5.3%</td>
</tr>
<tr>
<td>MicroBlaze</td>
<td>3.2%</td>
<td>1.3%</td>
<td>5.0%</td>
<td>0.7%</td>
</tr>
</tbody>
</table>
Many FGPU with different settings were tested

- Clock frequencies were fixed at 200MHz/400MHz
- BRAMs and DSPs were targeted but without the manufacturer’s IP-cores
- More FFs than LUTs were consumed due to the deep pipeline

- Empty pipeline stages were inserted between the two clock domains to improve timing

<table>
<thead>
<tr>
<th>Module</th>
<th>Parameter</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>CU</td>
<td># CUs</td>
<td>2, 4, 8</td>
</tr>
<tr>
<td></td>
<td># Outstanding mem. requests</td>
<td>16, 24, 32</td>
</tr>
<tr>
<td>Memory Controller</td>
<td>Cache Size</td>
<td>1 to 8KB</td>
</tr>
<tr>
<td></td>
<td># Cache Read Banks</td>
<td>2, 4, 8</td>
</tr>
<tr>
<td></td>
<td># Outstanding mem. requests</td>
<td>32, 64</td>
</tr>
<tr>
<td></td>
<td># AXI4 interfaces</td>
<td>1, 2, 4</td>
</tr>
<tr>
<td></td>
<td># Tag Managers</td>
<td>2, 4, 8, 16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Module</th>
<th>Parameter</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Controller</td>
<td>Cache Size</td>
<td>1 to 8KB</td>
</tr>
<tr>
<td></td>
<td># Cache Read Banks</td>
<td>2, 4, 8</td>
</tr>
<tr>
<td></td>
<td># Outstanding mem. requests</td>
<td>32, 64</td>
</tr>
<tr>
<td></td>
<td># AXI4 interfaces</td>
<td>1, 2, 4</td>
</tr>
<tr>
<td></td>
<td># Tag Managers</td>
<td>2, 4, 8, 16</td>
</tr>
</tbody>
</table>

Area requirements for different configurations

<table>
<thead>
<tr>
<th></th>
<th>LUTs</th>
<th>FFs</th>
<th>BRAM</th>
<th>DSPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available</td>
<td>219K</td>
<td>437K</td>
<td>545</td>
<td>900</td>
</tr>
<tr>
<td>8 CUs</td>
<td>57%</td>
<td>36%</td>
<td>31%</td>
<td>21%</td>
</tr>
<tr>
<td>4 CUs</td>
<td>35%</td>
<td>20%</td>
<td>18%</td>
<td>11%</td>
</tr>
<tr>
<td>2 CUs</td>
<td>26%</td>
<td>14%</td>
<td>12%</td>
<td>5.3%</td>
</tr>
<tr>
<td>2 CUs(min)</td>
<td>9.6%</td>
<td>8.1%</td>
<td>8.5%</td>
<td>5.3%</td>
</tr>
<tr>
<td>MicroBlaze</td>
<td>3.2%</td>
<td>1.3%</td>
<td>5.0%</td>
<td>0.7%</td>
</tr>
</tbody>
</table>
Many FGPUs with different settings were tested.

Clock frequencies were fixed at 200MHz/400MHz.

BRAMs and DSPs were targeted but without the manufacturer’s IP-cores.

More FFs than LUTs were consumed due to the deep pipeline.

Empty pipeline stages were inserted between the two clock domains to improve timing.

<table>
<thead>
<tr>
<th>Module</th>
<th>Parameter</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>CU</td>
<td># CUs</td>
<td>2, 4, 8</td>
</tr>
<tr>
<td></td>
<td># Outstanding mem. requests</td>
<td>16, 24, 32</td>
</tr>
<tr>
<td>Memory Controller</td>
<td>Cache Size</td>
<td>1 to 8KB</td>
</tr>
<tr>
<td></td>
<td># Cache Read Banks</td>
<td>2, 4, 8</td>
</tr>
<tr>
<td></td>
<td># Outstanding mem. requests</td>
<td>32, 64</td>
</tr>
<tr>
<td></td>
<td># AXI4 interfaces</td>
<td>1, 2, 4</td>
</tr>
<tr>
<td></td>
<td># Tag Managers</td>
<td>2, 4, 8, 16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Area requirements for different configurations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>Available</td>
</tr>
<tr>
<td>8 CUs</td>
</tr>
<tr>
<td>4 CUs</td>
</tr>
<tr>
<td>2 CUs</td>
</tr>
<tr>
<td>2 CUs(min)</td>
</tr>
<tr>
<td>MicroBlaze</td>
</tr>
</tbody>
</table>
Many FGPUs with different settings were tested.

Clock frequencies were fixed at 200MHz/400MHz.

BRAMs and DSPs were targeted but without the manufacturer’s IP-cores.

More FFs than LUTs were consumed due to the deep pipeline.

Empty pipeline stages were inserted between the two clock domains to improve timing.

<table>
<thead>
<tr>
<th>Module</th>
<th>Parameter</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>CU</td>
<td># CUs</td>
<td>2, 4, 8</td>
</tr>
<tr>
<td></td>
<td># Outstanding mem. requests</td>
<td>16, 24, 32</td>
</tr>
<tr>
<td>Memory Controller</td>
<td>Cache Size</td>
<td>1 to 8KB</td>
</tr>
<tr>
<td></td>
<td># Cache Read Banks</td>
<td>2, 4, 8</td>
</tr>
<tr>
<td></td>
<td># Outstanding mem. requests</td>
<td>32, 64</td>
</tr>
<tr>
<td></td>
<td># AXI4 interfaces</td>
<td>1, 2, 4</td>
</tr>
<tr>
<td></td>
<td># Tag Managers</td>
<td>2, 4, 8, 16</td>
</tr>
</tbody>
</table>

Area requirements for different configurations

<table>
<thead>
<tr>
<th></th>
<th>LUTs</th>
<th>FFs</th>
<th>BRAM</th>
<th>DSPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available</td>
<td>219K</td>
<td>437K</td>
<td>545</td>
<td>900</td>
</tr>
<tr>
<td>8 CUs</td>
<td>57%</td>
<td>36%</td>
<td>31%</td>
<td>21%</td>
</tr>
<tr>
<td>4 CUs</td>
<td>35%</td>
<td>20%</td>
<td>18%</td>
<td>11%</td>
</tr>
<tr>
<td>2 CUs</td>
<td>26%</td>
<td>14%</td>
<td>12%</td>
<td>5.3%</td>
</tr>
<tr>
<td>2 CUs(min)</td>
<td>9.6%</td>
<td>8.1%</td>
<td>8.5%</td>
<td>5.3%</td>
</tr>
<tr>
<td>MicroBlaze</td>
<td>3.2%</td>
<td>1.3%</td>
<td>5.0%</td>
<td>0.7%</td>
</tr>
</tbody>
</table>
Many FGPUs with different settings were tested.

Clock frequencies were fixed at 200MHz/400MHz.

BRAMs and DSPs were targeted but without the manufacturer’s IP-cores.

More FFs than LUTs were consumed due to the deep pipeline.

Empty pipeline stages were inserted between the two clock domains to improve timing.

### Adjustable parameters with direct influence on design scalability

<table>
<thead>
<tr>
<th>Module</th>
<th>Parameter</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>CU</td>
<td># CUs</td>
<td>2, 4, 8</td>
</tr>
<tr>
<td></td>
<td># Outstanding mem. requests</td>
<td>16, 24, 32</td>
</tr>
<tr>
<td>Memory</td>
<td>Cache Size</td>
<td>1 to 8KB</td>
</tr>
<tr>
<td>Controller</td>
<td># Outstanding mem. requests</td>
<td>32, 64</td>
</tr>
<tr>
<td></td>
<td># Cache Read Banks</td>
<td>2, 4, 8</td>
</tr>
<tr>
<td></td>
<td># AXI4 interfaces</td>
<td>1, 2, 4</td>
</tr>
<tr>
<td></td>
<td># Tag Managers</td>
<td>2, 4, 8, 16</td>
</tr>
</tbody>
</table>

### Area requirements for different configurations

<table>
<thead>
<tr>
<th></th>
<th>LUTs</th>
<th>FFs</th>
<th>BRAM</th>
<th>DSPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available</td>
<td>219K</td>
<td>437K</td>
<td>545</td>
<td>900</td>
</tr>
<tr>
<td>8 CUs</td>
<td>57%</td>
<td>36%</td>
<td>31%</td>
<td>21%</td>
</tr>
<tr>
<td>4 CUs</td>
<td>35%</td>
<td>20%</td>
<td>18%</td>
<td>11%</td>
</tr>
<tr>
<td>2 CUs</td>
<td>26%</td>
<td>14%</td>
<td>12%</td>
<td>5.3%</td>
</tr>
<tr>
<td>2 CUs (min)</td>
<td>9.6%</td>
<td>8.1%</td>
<td>8.5%</td>
<td>5.3%</td>
</tr>
<tr>
<td>MicroBlaze</td>
<td>3.2%</td>
<td>1.3%</td>
<td>5.0%</td>
<td>0.7%</td>
</tr>
</tbody>
</table>
Results
Area Requirements

- Many FGPUs with different settings were tested
- Clock frequencies were fixed at 200MHz/400MHz
- BRAMs and DSPs were targeted but without the manufacturer’s IP-cores
- More FFs than LUTs were consumed due to the deep pipeline
  - Empty pipeline stages were inserted between the two clock domains to improve timing

<table>
<thead>
<tr>
<th>Module</th>
<th>Parameter</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>CU</td>
<td># CUs</td>
<td>2, 4, 8</td>
</tr>
<tr>
<td></td>
<td># Outstanding mem. requests</td>
<td>16, 24, 32</td>
</tr>
<tr>
<td>Memory Controller</td>
<td>Cache Size</td>
<td>1 to 8KB</td>
</tr>
<tr>
<td></td>
<td># Cache Read Banks</td>
<td>2, 4, 8</td>
</tr>
<tr>
<td></td>
<td># Outstanding mem. requests</td>
<td>32, 64</td>
</tr>
<tr>
<td></td>
<td># AXI4 interfaces</td>
<td>1, 2, 4</td>
</tr>
<tr>
<td></td>
<td># Tag Managers</td>
<td>2, 4, 8, 16</td>
</tr>
</tbody>
</table>

Area requirements for different configurations

<table>
<thead>
<tr>
<th></th>
<th>LUTs</th>
<th>FFs</th>
<th>BRAM</th>
<th>DSPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available</td>
<td>219K</td>
<td>437K</td>
<td>545</td>
<td>900</td>
</tr>
<tr>
<td>8 CUs</td>
<td>57%</td>
<td>36%</td>
<td>31%</td>
<td>21%</td>
</tr>
<tr>
<td>4 CUs</td>
<td>35%</td>
<td>20%</td>
<td>18%</td>
<td>11%</td>
</tr>
<tr>
<td>2 CUs</td>
<td>26%</td>
<td>14%</td>
<td>12%</td>
<td>5.3%</td>
</tr>
<tr>
<td>2 CUs(min)</td>
<td>9.6%</td>
<td>8.1%</td>
<td>8.5%</td>
<td>5.3%</td>
</tr>
<tr>
<td>MicroBlaze</td>
<td>3.2%</td>
<td>1.3%</td>
<td>5.0%</td>
<td>0.7%</td>
</tr>
</tbody>
</table>
Results

Speedup over MicroBlaze

- The MicroBlaze is:
  - configured by default settings for best performance
  - clocked at 185MHz
- 49x average speedup was achieved overall
- 166x maximum speedup for matrix multiplication
- 7.7 Gbps maximum throughput when used like a DMA (memcpy), averaged over the whole execution time

Wall clock time speedup for 8 CUs over MicroBlaze implementation for task size between 256 and 256K

FGPU | FPGA’16, Monterey, 23 February 2016
The MicroBlaze is:
- configured by default settings for best performance
- clocked at 185MHz
- 49x average speedup was achieved overall
- 166x maximum speedup for matrix multiplication
- 7.7 Gbps maximum throughput when used like a DMA (memcopy), averaged over the whole execution time

Wall clock time speedup for 8 CUs over MicroBlaze implementation for task size between 256 and 256K

FGPU | FPGA’16, Monterey, 23 February 2016
Results

Speedup over MicroBlaze

- The MicroBlaze is:
  - configured by default settings for best performance
  - clocked at 185MHz
- 49x average speedup was achieved overall
- 166x maximum speedup for matrix multiplication
- 7.7 Gbps maximum throughput when used like a DMA (memcpy), averaged over the whole execution time

Wall clock time speedup for 8 CUs over MicroBlaze implementation for task size between 256 and 256K
The MicroBlaze is:
- configured by default settings for best performance
- clocked at 185MHz

49x average speedup was achieved overall

166x maximum speedup for matrix multiplication

7.7 Gbps maximum throughput when used like a DMA (memcpy), averaged over the whole execution time
Results

Speedup over MicroBlaze

- The MicroBlaze is:
  - configured by default settings for best performance
  - clocked at 185MHz
- 49x average speedup was achieved overall
- 166x maximum speedup for matrix multiplication
- 7.7 Gbps maximum throughput when used like a DMA (memcopy), averaged over the whole execution time

Wall clock time speedup for 8 CUs over MicroBlaze implementation for task size between 256 and 256K
The MicroBlaze is:
- configured by default settings for best performance
- clocked at 185MHz

49x average speedup was achieved overall

166x maximum speedup for matrix multiplication

7.7 Gbps maximum throughput when used like a DMA (memcopy), averaged over the whole execution time

Wall clock time speedup for 8 CUs over MicroBlaze implementation for task size between 256 and 256K
The ARM CPU is:
- supported by the NEON vector engine
- clocked at 667MHz

3.5x average speedup was achieved overall

35x maximum speedup for matrix multiplication

Better speedups were recorded for more computational intensive tasks

Cache flushing on ARM was done only for the dirty region

Wall clock time speedup for 8 CUs over ARM+NEON implementation for task size between 256 and 256K
Results

Speedup over ARM+NEON

- The ARM CPU is:
  - supported by the NEON vector engine
  - clocked at 667MHz
- 3.5x average speedup was achieved overall
- 35x maximum speedup for matrix multiplication
- Better speedups were recorded for more computational intensive tasks
- Cache flushing on ARM was done only for the dirty region

Wall clock time speedup for 8 CUs over ARM+NEON implementation for task size between 256 and 256K
The ARM CPU is:
- supported by the NEON vector engine
- clocked at 667MHz

3.5x average speedup was achieved overall
35x maximum speedup for matrix multiplication
Better speedups were recorded for more computational intensive tasks
Cache flushing on ARM was done only for the dirty region

Wall clock time speedup for 8 CUs over ARM+NEON implementation for task size between 256 and 256K

FGPU | FPGA’16, Monterey, 23 February 2016
The ARM CPU is:
- supported by the NEON vector engine
- clocked at 667MHz

3.5x average speedup was achieved overall

35x maximum speedup for matrix multiplication

Better speedups were recorded for more computational intensive tasks

Cache flushing on ARM was done only for the dirty region
Results
Speedup over ARM+NEON

- The ARM CPU is:
  - supported by the NEON vector engine
  - clocked at 667MHz
- 3.5x average speedup was achieved overall
- 35x maximum speedup for matrix multiplication
- Better speedups were recorded for more computational intensive tasks
- Cache flushing on ARM was done only for the dirty region

Wall clock time speedup for 8 CUs over ARM+NEON implementation for task size between 256 and 256K
Results

Speedup over ARM+NEON

- The ARM CPU is:
  - supported by the NEON vector engine
  - clocked at 667MHz

- 3.5x average speedup was achieved overall

- 35x maximum speedup for matrix multiplication

- Better speedups were recorded for more computational intensive tasks

- Cache flushing on ARM was done only for the dirty region

Wall clock time speedup for 8 CUs over ARM+NEON implementation for task size between 256 and 256K
Results

Speedup over ARM+NEON

- The ARM CPU is:
  - supported by the NEON vector engine
  - clocked at 667MHz
- 3.5x average speedup was achieved overall
- 35x maximum speedup for matrix multiplication
- Better speedups were recorded for more computational intensive tasks
- Cache flushing on ARM was done only for the dirty region

Wall clock time speedup for 8 CUs over ARM+NEON implementation for task size between 256 and 256K
Better speedups achieved when processing bigger pieces of data.

The speedup increases more rapidly for the most complex applications.

A min. of 4us is needed by FGPU for:
- preparing to execute a new task at the beginning;
- flushing the cache content at the end.

Wall clock time speedup for 8 CUs over ARM+NEON implementation for variable task size.
Better speedups achieved when processing bigger pieces of data

The speedup increases more rapidly for the most complex applications

A min. of 4us is needed by FGPU for:
- preparing to execute a new task at the beginning
- flushing the cache content at the end
Better speedups achieved when processing bigger pieces of data.

The speedup increases more rapidly for the most complex applications.

A minimum of 4us is needed by FGPU for:

- preparing to execute a new task at the beginning
- flushing the cache content at the end

Wall clock time speedup for 8 CUs over ARM+NEON implementation for variable task size.
Better speedups achieved when processing bigger pieces of data

The speedup increases more rapidly for the most complex applications

A min. of 4us is needed by FGPU for:
- preparing to execute a new task at the beginning
- flushing the cache content at the end

Wall clock time speedup for 8 CUs over ARM+NEON implementation for variable task size
Better speedups achieved when processing bigger pieces of data.

The speedup increases more rapidly for the most complex applications.

A min. of 4us is needed by FGPU for:
- preparing to execute a new task at the beginning
- flushing the cache content at the end

Wall clock time speedup for 8 CUs over ARM+NEON implementation for variable task size.
Better speedups achieved when processing bigger pieces of data

- The speedup increases more rapidly for the most complex applications
- A min. of 4us is needed by FGPU for:
  - preparing to execute a new task at the beginning
  - flushing the cache content at the end
Results

Number of CUs

- Implementing more CUs improved always the speedups
  - linearly in best cases
  - even for less computationally intensive applications
- Having multiple CUs improves the efficiency of the memory controller
  - by pushing more memory requests in the pipeline

Wall clock time speedup for FGPU with different number of CUs over MicroBlaze implementation for a task size of 16K
Results

Number of CUs

- Implementing more CUs improved always the speedups
  - linearly in best cases
  - even for less computationally intensive applications
- Having multiple CUs improves the efficiency of the memory controller
  - by pushing more memory requests in the pipeline

Wall clock time speedup for FGPU with different number of CUs over MicroBlaze implementation for a task size of 16K

 FGPU | FPGA’16, Monterey, 23 February 2016
Results

Number of CUs

- Implementing more CUs improved always the speedups
  - linearly in best cases
  - even for less computationally intensive applications
- Having multiple CUs improves the efficiency of the memory controller
  - by pushing more memory requests in the pipeline
Implementing more CUs improved always the speedups
- linearly in best cases
- even for less computationally intensive applications

Having multiple CUs improves the efficiency of the memory controller
- by pushing more memory requests in the pipeline

Wall clock time speedup for FGPU with different number of CUs over MicroBlaze implementation for a task size of 16K
Results
Number of CUs

- Implementing more CUs improved always the speedups
  - linearly in best cases
  - even for less computationally intensive applications
- Having multiple CUs improves the efficiency of the memory controller
  - by pushing more memory requests in the pipeline

Wall clock time speedup for FGPU with different number of CUs over MicroBlaze implementation for a task size of 16K
Results

Power Consumption

- The measurements were done using the on-board power monitor.
- The 2\textsuperscript{nd} ARM core was recording the power consumption while the 1\textsuperscript{st} was controlling FGPU.
- In average over the whole benchmark:
  - The biggest and smallest FGPU\textsuperscript{s} consumed 13x and 3.3x more power than the MicroBlaze.
  - 5.2W was recorded as maximum by the power monitor.
The measurements were done using the on-board power monitor.

The 2\textsuperscript{nd} ARM core was recording the power consumption while the 1\textsuperscript{st} was controlling FGPU.

In average over the whole benchmark:

- The biggest and smallest FGPs consumed 13x and 3.3x more power than the MicroBlaze.
- 5.2W was recorded as maximum by the power monitor.
Results

Power Consumption

- The measurements were done using the on-board power monitor.
- The 2\textsuperscript{nd} ARM core was recording the power consumption while the 1\textsuperscript{st} was controlling FGPU.
- In average over the whole benchmark:
  - The biggest and smallest FGPsUs consumed 13x and 3.3x more power than the MicroBlaze.
  - 5.2W was recorded as maximum by the power monitor.
Results
Power Consumption

- The measurements were done using the on-board power monitor.
- The 2nd ARM core was recording the power consumption while the 1st was controlling FGPU.
- In average over the whole benchmark:
  - The biggest and smallest FGPUs consumed 13x and 3.3x more power than the MicroBlaze.
- 5.2W was recorded as maximum by the power monitor.
Results
Power Consumption

- The measurements were done using the on-board power monitor.
- The 2\textsuperscript{nd} ARM core was recording the power consumption while the 1\textsuperscript{st} was controlling FGPU.
- In average over the whole benchmark:
  - The biggest and smallest FGPs consumed 13x and 3.3x more power than the MicroBlaze.
  - 5.2W was recorded as maximum by the power monitor.
Results

Summary

In comparison to the MicroBlaze, all tested FGPUs:
- consume 3.2x - 4.4x less energy
- speedup execution by 11x - 49x
- need 3.0x - 17.7x more area

In comparison to ARM+NEON, all tested FGPUs:
- speedup execution by up to 3.5x

Average wall clock time speedup, power saving and area overhead for different FGPUs over MicroBlaze and ARM+NEON implementations. Speedups were averaged over the whole benchmark and problem sizes from 256 to 256K.
Results

Summary

- In comparison to the MicroBlaze, all tested FGPUs:
  - consume 3.2x - 4.4x less energy
  - speedup execution by 11x - 49x
  - need 3.0x - 17.7x more area

- In comparison to ARM+NEON, all tested FGPUs:
  - speedup execution by up to 3.5x
Results

Summary

- In comparison to the MicroBlaze, all tested FGPUs:
  - consume 3.2x - 4.4x less energy
  - speedup execution by 11x - 49x
  - need 3.0x - 17.7x more area

- In comparison to ARM+NEON, all tested FGPUs:
  - speedup execution by up to 3.5x

Average wall clock time speedup, power saving and area overhead for different FGPUs over MicroBlaze and ARM+NEON implementations. Speedups were averaged over the whole benchmark and problem sizes from 256 to 256K.
In comparison to the MicroBlaze, all tested FGPUs:

- consume 3.2x - 4.4x less energy
- speedup execution by 11x - 49x
- need 3.0x - 17.7x more area

In comparison to ARM+NEON, all tested FGPUs:

- speedup execution by up to 3.5x

Average wall clock time speedup, power saving and area overhead for different FGPUs over MicroBlaze and ARM+NEON implementations. Speedups were averaged over the whole benchmark and problem sizes from 256 to 256K.
Results

Summary

- In comparison to the MicroBlaze, all tested FGPFUs:
  - consume 3.2x - 4.4x less energy
  - speed up execution by 11x - 49x
  - need 3.0x - 17.7x more area

- In comparison to ARM+NEON, all tested FGPFUs:
  - speed up execution by up to 3.5x

Average wall clock time speedup, power saving and area overhead for different FGPFUs over MicroBlaze and ARM+NEON implementations. Speedups were averaged over the whole benchmark and problem sizes from 256 to 256K.
Outline

1 Motivation and Background

2 FGPU Architecture
   - Execution Model
   - Platform Model
   - Compute Unit Architecture
   - Global Memory Controller

3 Implementation and Results

4 Future Work and Conclusion
Future Work

- Extend the ISA to cover more benchmarks
- Developing an LLVM-backend to compile OpenCL-kernels
- Developing a Linux-driver with OpenCL-compatible interface
- Enabling branches at the work-item level
- Supporting soft/hard floating point computations
- Providing local/global atomic operations
- Improving the cache system by having two levels
- Implementing local storage within the CUs to be compliant to the OpenCL memory model.
Future Work

- Extend the ISA to cover more benchmarks
- Developing an LLVM-backend to compile OpenCL-kernels
- Developing a Linux-driver with OpenCL-compatible interface
- Enabling branches at the work-item level
- Supporting soft/hard floating point computations
- Providing local/global atomic operations
- Improving the cache system by having two levels
- Implementing local storage within the CUs to be compliant to the OpenCL memory model.
Future Work

- Extend the ISA to cover more benchmarks
- Developing an LLVM-backend to compile OpenCL-kernels
- Developing a Linux-driver with OpenCL-compatible interface
- Enabling branches at the work-item level
- Supporting soft/hard floating point computations
- Providing local/global atomic operations
- Improving the cache system by having two levels
- Implementing local storage within the CUs to be compliant to the OpenCL memory model.
Future Work

- Extend the ISA to cover more benchmarks
- Developing an LLVM-backend to compile OpenCL-kernels
- Developing a Linux-driver with OpenCL-compatible interface
  - Enabling branches at the work-item level
  - Supporting soft/hard floating point computations
  - Providing local/global atomic operations
  - Improving the cache system by having two levels
  - Implementing local storage within the CUs to be compliant to the OpenCL memory model.
Future Work

- Extend the ISA to cover more benchmarks
- Developing an LLVM-backend to compile OpenCL-kernels
- Developing a Linux-driver with OpenCL-compatible interface
- Enabling branches at the work-item level
- Supporting soft/hard floating point computations
- Providing local/global atomic operations
- Improving the cache system by having two levels
- Implementing local storage within the CUs to be compliant to the OpenCL memory model.
Future Work

- Extend the ISA to cover more benchmarks
- Developing an LLVM-backend to compile OpenCL-kernels
- Developing a Linux-driver with OpenCL-compatible interface
- Enabling branches at the work-item level
- Supporting soft/hard floating point computations
  - Providing local/global atomic operations
  - Improving the cache system by having two levels
  - Implementing local storage within the CUs to be compliant to the OpenCL memory model.
Future Work

- Extend the ISA to cover more benchmarks
- Developing an LLVM-backend to compile OpenCL-kernels
- Developing a Linux-driver with OpenCL-compatible interface
- Enabling branches at the work-item level
- Supporting soft/hard floating point computations
- Providing local/global atomic operations
- Improving the cache system by having two levels
- Implementing local storage within the CUs to be compliant to the OpenCL memory model.
Future Work

- Extend the ISA to cover more benchmarks
- Developing an LLVM-backend to compile OpenCL-kernels
- Developing a Linux-driver with OpenCL-compatible interface
- Enabling branches at the work-item level
- Supporting soft/hard floating point computations
- Providing local/global atomic operations
- Improving the cache system by having two levels
- Implementing local storage within the CUs to be compliant to the OpenCL memory model.
Future Work

- Extend the ISA to cover more benchmarks
- Developing an LLVM-backend to compile OpenCL-kernels
- Developing a Linux-driver with OpenCL-compatible interface
- Enabling branches at the work-item level
- Supporting soft/hard floating point computations
- Providing local/global atomic operations
- Improving the cache system by having two levels
- Implementing local storage within the CUs to be compliant to the OpenCL memory model.
Thank you for your attention!
Instruction Set Architecture

Example: FIR Filter

__kernel void fir(
    int *input_array,
    int *filter,
    int *res_array,
    int filter_len)
{
    int index = get_global_id(0);
    int i = 0;
    int acc = 0;

    do
    {
        acc += input_array[index+i] * filter[i];
        i++;
    } while(i != filter_len);

    res[index] = acc;
}

(a) FIR filter as OpenCL kernel

# FIR filter using 1D index space. It has 4 Parameters:
# 0 : address of the first element in input array
# 1 : address of the first element in coefficients array
# 2 : address of the first element in results array
# 3 : filter length (L)
LID r1, d0 # local ID: load the local work-item index in its work-group into r1
WGOFF r2, d0 # Work-Group Offset: load the work-group global offset into r2
ADD r1, r1, r2 # ADD integers: r1 has now the global id of the work-item
LP r2, 3 # Load Parameter: r2 has filter length
LP r3, 0 # Load Parameter: r3 is a pointer to the input array
LP r4, 1 # Load Parameter: r4 is a pointer to the coefficients array
ADDI r5, r0, 0 # ADD Immediate: r5 will be the loop index (initialized with 0)
ADDI r6, r0, 0 # ADD Immediate: r6 will contain the result (initialized with 0)

begin: LW r10, r4[r5] # Load Word: load a coefficients into r10
ADD r11, r5, r1 # ADD integers: calculate the index of an element in input array
LW r11, r3[r11] # Load Word: load the input element into r11
MACC r6, r10, r11 # Multiply and ACCumulate: update the result
ADDI r5, r5, 1 # ADD Immediate: update loop index
BNE r5, r2, begin # Branch if Not Equal: repeat the iteration if necessary

LP r20, 2 # Load Parameter: r20 is a pointer to the result array
SW r6, r20[r1] # Store Word: store the result r6 into the index r1 in result array
RET # RETurn: end of task

(b) Equivalent implementation in FGPU ISA