From JVM to FPGA: Bridging Abstraction Hierarchy via Optimized Deep Pipelining

Jason Cong, Peng Wei and Cody Hao Yu

University of California, Los Angeles
Motivation:

Harnessing FPGAs in Datacenter
Harnessing FPGAs in Datacenters: Why?

- Heterogeneous architecture: an “agreement” from the hardware community

![Graph showing energy efficiency and processor number](image)

Generalization ► Customization

Sources: Google, NVIDIA, Intel, Bob Broderson, Berkeley Wireless group
Harnessing FPGAs in Datacenters: Why?

- Heterogeneous architecture: an “agreement” from the hardware community
- The FPGA-based cluster is a promising paradigm
  - Standalone FPGA accelerators demonstrate orders-of-magnitude performance/watt improvement
Harnessing FPGAs in Datacenters: Why?

- Heterogeneous architecture: an "agreement" from the hardware community

- The FPGA-based cluster is a promising paradigm
  - Standalone FPGA accelerators demonstrate orders-of-magnitude performance/watt improvement
  - FPGAs are reconfigurable:
    - A relatively "general" specialized device
    - It is now in the cloud
Challenge:

system integration - from kernel speedup to system acceleration
Accelerator (FPGA)-as-a-Service

Global Accelerator Manager: accelerator-centric scheduling

Node Accelerator Manager: local accelerator service management, JVM-to-ACC communication optimization
In terms of performance, there is much to say...

- **Time breakdown of AES**
  - Pack: app.-dep.; ~4GB/s
  - Send (via socket): ~3GB/s
  - Usr->Kernel: ~6GB/s
  - DMA: ~5GB/s
  - Load: ~6GB/s (shd w/ Store)
  - Compute: 12.8GB/s
    - >100x over CPU
  - ...
  - \( \frac{1}{1/4 + 1/3 + \ldots} = 0.47 \text{ GB/s} \)
  - **27x performance loss!!!**
JVM-FPGA Communication Pipelining

- **Java**
  - Input Java Objects
  - Pack
  - Send
  - Pageable Input Buffer Queue
  - Pin
  - DMA Forward
  - Device Input Buffer Queue
  - Load
  - Compute

- **C/C++ & Vendor APIs**
  - Pageable Output Buffer Queue
  - Unpin
  - DMA Backward
  - Device Output Buffer Queue
  - Store

- **Verilog HDL**
  - BRAM Output Buffer Queue
  - BRAM Input Buffer Queue

- **Java Objects**
  - Load
  - Store

- **BRAM**
  - Load
  - Store
**JVM-FPGA Communication Pipelining**

- **Send + Pin => Send**
  - Limitation of vendor APIs
- **Load/Compute/Store => Compute**
  - Overlapping comm. & comp.
- **Programmer’s responsibility**
  - Pack and unpack
  - Implementing an iterator interface to supply input data
  - Header + payload
In terms of performance, there is still much to say...

- The pipeline efficiency is bounded by the slowest stage
- In general, \( \text{latency} = \text{time\_setup} \ (\text{one\_time}) + \text{payload\_size} \times \text{time\_unit} \ (\text{linear}) \)
- Adjust the payload sizes of different pipeline stages to balance their throughputs
- ... but how to?
- Linear with constraints => **linear programming**
Linear Programming Formulation

**Problem Formulation:**
maximize the pipeline throughput, i.e.,

\[ T_K = \max (T_{\text{pack}}, T_{\text{send}}, \ldots, T_{\text{unpack}}) \]

\[ T_{\text{stage}} = \frac{1}{L_{\text{stage}}} = \frac{1}{f_{\text{stage}}(S_{\text{stage}})} \]

**Modeling of Data Transfer Stages:**
for each individual data transfer stage, impose the payload size constraint, and model the relation between the payload size and the latency via linear equations:

\[ L_{\text{stage}} = L_{\text{setup}} + S_{\text{stage}} \times L_{\text{unit}} \]

\[ S_{\text{stage}} \leq S_{\text{max}}^{\text{stage}} \]

**Modeling of Compute Stage:**
profile a set of payload sizes (power of two), and model the latency of the compute stage into a selection equation with a set of binary variables:

\[ L_{\text{compute}} = \sum_i p_i \times L_{S_i}, \text{ where } \sum_i p_i = 1, \; p_i \in \{0, 1\} \]

**Modeling of Memory Constraints:**
constrain the memory usage of the pipeline in both the CPU and the FPGA sides for separate-memory platforms, and in only the CPU side for shared-memory platforms:

\[ \sum S_{Q_{\text{stage}}} = \sum (S_{\text{stage}} \times D_{\text{stage}}) \leq S_{\text{capacity}} \]
Experimental Results

- A set of computation kernels as benchmarks
- Each with a Java program as the host
- Currently single-threaded, and will showcase the real-application results in the near future
Experimental Results

- A set of computation kernels as benchmarks
- Each with a Java program as the host
- Currently single-threaded, and will showcase the real-application results in the near future
Lessons Learned and Future Work

- Single thread -> multiple threads -> Mainstream frameworks
  - Modeling in the multithreaded scenario
  - Integration with frameworks like Apache Hadoop and Spark

- Adapt to various platforms
  - Latest platforms support FPGA’s direct access of user-space data, like IBM CAPI and Intel Xeon+FPGA
    - Amazon EC2 F1 instance brings virtualization into consideration

- JVM related improvement
  - Fast and safe allocation and management of native-space memory
THANKS FOR YOUR ATTENTION.
Discussion

Problem Formulation:
maximize the pipeline throughput, i.e.,
\[ T_K = \min(T_{pack}, T_{send}, \ldots, T_{unpack}) \]
\[ T_{stage} = \frac{1}{L_{stage}} = \frac{1}{f_{stage}(S_{stage})} \]

Modeling of Data Transfer Stages:
for each individual data transfer stage, impose the payload size constraint, and model the relation between the payload size and the latency via linear equations:
\[ L_{stage} = L_{stage}^{\text{min}} + S_{stage} \times L_{stage}^{\text{unit}} \]
\[ S_{stage} \leq S_{stage}^{\text{max}} \]

Modeling of Compute Stage:
profile a set of payload sizes (power of two), and model the latency of the compute stage into a selection equation with a set of binary variables:
\[ L_{compute} = \sum p_i \times L_{S_i}, \text{ where } \sum p_i = 1, p_i \in \{0,1\} \]

Modeling of Memory Constraints:
constrain the memory usage of the pipeline in both the CPU and the FPGA sides for separate-memory platforms, and in only the CPU side for shared-memory platforms:
\[ \sum S_{D_{stage}} = \sum (S_{stage} \times D_{stage}) \leq S_{capacity} \]