Streaming Similarity Search on FPGA based on Dynamic Time Warping

Yu WANG
Associate Prof.,
Head, Research Institution of Circuits and Systems,
E.E. Dept, Tsinghua University, Beijing, China
http://nics.ee.tsinghua.edu.cn/people/wangyu/
Joint work by Tsinghua Univ. and IBM China Research Lab
Based on a submitted paper to FPGA 2013

Nano-scale Integrated Circuit and System Lab.
Outline

- Background and Motivation
  - Why we need streaming similarity search
  - Recent achievements and problems to solve
- Subsequence Similarity Search on FPGA
  - Algorithms
  - Hardware Architectures
  - Results
- Conclusion and future work
Vision 2025

- Every object will be smart
- The Ensemble is the Function!
  - Function determined by availability of sensing, actuation, connectivity, computation, storage and energy
  - Collaborating to present unifying experiences or to fulfill common goals

A humongous networked, distributed, adaptive hierarchical, hybrid control problem

Something is happening... CYBER PHYSICAL SYSTEMS, INTERNET OF THINGS, SYSTEMS OF SYSTEMS...!

INSTRUMENTED
We now have the ability to measure, sense and see the exact condition of practically everything.

INTERCONNECTED
People, systems and objects can communicate and interact with each other in entirely new ways.

INTELLIGENT
We can respond to changes quickly and accurately and get better results by predicting and optimizing for future events.
Internet of Things

### Nowadays
- Independent Applications
- Traditional Database Techniques
- Small Scale “small IOTs”
- Monitoring only

### Future
- Fully connected, and correlated Applications
- Advanced IT techniques
- Large Scale, and large Volume Data (“big IOTs”)  
- Different realtime or non-realtime applications

**BIG DATA (Time and Spatial Correlated Streaming DATA)**
Volume, Variety, Velocity

**Collection, Publish, Processing, Storage, and Query for BIG DATA**

**IoT DATA Manage System (IBM RODB©)**
RODB

Different Applications

Application Specific Data Management Middleware (Collection, Publish, Processing, Storage, and Query)

Realtime Oriented Database
Data format from IOT (CPS, SoS, etc.)

**Format of Data**
- RFID

**Industries in Smarter Planet**
- Logistic
- Retail
- Mineral
- Steel
- Manufactory
- E&U
- Petro
- Chemistry
- Smart building
- Smart City
- Environment monitoring
- Healthcare
- Transportation
Mining Task Dependency (Not Complete)

- No history data involved
- May have real time req
- History data analyses

- Similarity Search
- Segmentation
- Correlation Discovery
- Motif Discovery
- Visualization

- Classification
- Clustering

- Data Privacy
- Prediction
- Burst Detection

- Novelty/Anomaly detection
- Rule Discovery
“Similarity” Search

- Finite field subsequence exact search
  - Object: string
  - e.g. find “pattern” in “we have a pattern here” with K.M.P

- Finite field subsequence similarity search
  - Object: DNA chain, Protein sequence
  - e.g. find similar subsequence as “ATGAG” in a DNA chain “ATGACTGAG…” with Smith-Waterman.

- Infinite field subsequence similarity search
  - Object: time series data
  - e.g. next slide
Streaming Subsequence Similarity Search

- Time series (electrocardiogram) & pattern (query)
- Pick out subsequences with sliding window (totally N subsequences)
- Compare the subsequences with the pattern, under a certain distance measure, to judge if they are similar

Simple DATA representation Tuple [Sensor, Time, Value]

Time Complexity $O(N \times O(distance))$
Distance Measure

- **Dynamic Time Warping**

\[ P = p_1, p_2, p_3 \ldots p_M \; ; \; S = s_1, s_2, s_3 \ldots s_M \]

\[ DTW(S, P) = D(M, M) \]

\[
D(i, j) = \text{dist}(s_i, p_j) + \min \left\{ D(i-1, j), D(i, j-1), D(i-1, j-1) \right\}
\]

- **Step 1:** Calculate the distance of each two points

- **Step 2:** Find the shortest accumulated path

- **DTW** is the best distance measure in most domains. It allows shrinking, sketching, warping, even different lengths. Distance Complexity Analysis (\(O(M \times M)\))
Challenges for Streaming Similarity Computing

- **Challenges (Velocity, Volume, Variety)**
  - **Real time Analysis**
    - Both on the sensor part or cloud
  - **Large Volume Streaming DATA to be compared**
    - Can not afford to storage on Sensors
    - Millions of Sensors may be on the Edges
  - **Various Patterns**
    - People may want to search for different patterns on different/same dataset

- **Previous Work**
  - Software preprocessing to reduce the real DTW
  - Parallel Hardware: More on task level parallelism, little was performed for fine-grained parallelism
1000+ papers on software speedup techniques:

1. Y. Sakurai et al. proposed a computation-reuse algorithm called SPRING [3]

- Only one tuple is different between the two neighboring subsequences
- Merge N M-by-M matrixes into single N-by-M matrix. N paths grow at the same time

- It reduces the time complexity from $O(N^*M^*M)$ to $O(N^*M)$

The whole sequence can’t be normalized in streaming:
2. Lower bound: A. Fu, E. Keogh et al. tried to estimate the lower bound of DTW distance in a cheap way, called LB_Keogh [1].

- It constrains the warping path will not deviate more than R*M cells from the diagonal. Generate an upper envelope and a lower envelope, and the sum of the subsequence not falling within the bounding envelope is defined as the LB_Keogh.

- If the lower bound distance exceeds the threshold, the DTW distance will also exceed the threshold, and then the subsequence can be pruned off.
3. S. H. Lim et al. used indexing techniques to speed up the search [11]
   - Build a look up table for different patterns; subsequences searching speed equals to the look up table searching speed, which is very fast
   - Look up table construction cost is even larger than DTW, only suitable for frequent querying in the same sequence.
   - No one can index on a streaming sequence which may be infinitely long.

4. There are also some other techniques, like early abandoning.
Several works try to exploit parallel hardware, such as multi-cores[8], computer cluster[6], GPU[4] to speedup the search.

- All these works try to allocate subsequences starting from different position of the whole sequence to different processing units, which can be seen as coarse-grained parallelism.
- [4] also uses threads to parallel generate the warping matrixes, but serially does the path searching, which can be seen as partial fine-grained parallelism.
- Lead to a heavy data-transfer burden, as one subsequence may consist of too many tuples. The [4]'s partial fine-grained parallel work even needs to transfer a whole matrix between thread.

Related Work -- Parallel Hardware
The first and only work[2] using FPGA is generated by a C-to-VHDL tool called ROCCC.

- From the reported performance, we think the tool exploits the fine-grained parallelism inside DTW.
- It does not exploit the coarse-grained parallelism.
- The lack of insight into FPGA limits the scalability and flexibility:
  - It can not support patterns of length larger than 128.
  - It can not support on-line updating patterns of different lengths. For example, if a new pattern of length 127 is wanted, it must re-compile the system and re-download the FPGA, which may cost several hours.

Related Work -- Parallel Hardware
Problems we try to solve

Problems
- Software can’t accelerate DTW itself
- Coarse-grained parallelism may leads to heavy burden on bandwidth
- Fine-grained parallelism requires hard-wired synchronization
- FPGA lacks flexibility as software

Solutions
- Turn to parallel hardware to accelerate DTW
- Choose and modify streaming parallel algorithms (SPRING) to reduce bandwidth
- Use FPGA with flexible structure for fine grained parallelism
Outline

- Background and Motivation
  - Why we need streaming similarity search
  - Recent achievements and problems to solve

- Subsequence Similarity Search on FPGA
  - Algorithms
  - Hardware Architectures
  - Results

- Conclusion and future work
Algorithms

- Normalization
  - Enable multiple DTW
- Hybrid lower bound
  - Good Preprocessing to leave very few real DTW
- Multiple DTW
  - Coarse-Grain and Fine-Grain Parallelism

Algorithm Framework

Normalizer → Hybrid Lower Bound → Multiple DTW
Normalization

Assumption: the offset or the amplitude can be approximately seen as time-invariant in a little longer length of M+C, where M is the length of pattern, and C is a constant.
Hybrid Lower Bound

- **LB_partial DTW**
  - Stable but time-consuming

- **LB_Keogh/ reversed LB_Keogh**
  - Efficient but in-stable
  - Significantly degrade when $R$ increase

$$U_i = \max \{ P_{i-R}, P_{i-R+1} \ldots P_{i+R}, P_{i+R} \};$$

$$L_i = \min \{ P_{i-R}, P_{i-R+1} \ldots P_{i+R}, P_{i+R} \};$$

$$D_i = \begin{cases} 
S_i - U_i & \text{if } S_i > U_i \\
L_i - S_i & \text{if } L_i > S_i \\
0 & \text{else}
\end{cases}$$

$$LB(P_{1,Y}, S_{1,Y}) = \sum \{D_1, D_2, \ldots, D_Y\}$$

This can been seen as a combination of early abandoning technique and lower bounding technique.
Subsequences that have not been pruned off.
Multiple DTW – Modified SPRING

SPRING:

SPRING: $C^*$

$DTW(S_{s,e}, P) = D(e,M)$

$D(i, j) = dist(s_i, p_j) + \min$

- $i-R < sp(i-1, j) + j < i+R$ \(\Rightarrow D(i-1, j) : INF \)
- $i-R < sp(i-1, j-1) + j < i+R$ \(\Rightarrow D(i-1, j-1) : INF \)
- $i-R < sp(i, j) + j < i+R$ \(\Rightarrow D(i, j) : INF \)

$D(i, 0) = \begin{cases} 0, & \text{if valid}(i) == 1; \\ INF, & \text{if valid}(i) == 0; \end{cases}$

$D(0, j) = infinite; where 1<i<N, 1<j<M.$

$Sp(i, j) = \begin{cases} Sp(i-1, j), & \text{if } D(i-1, j) \text{ is the minimum} \\ Sp(i-1, j-1), & \text{if } D(i-1, j-1) \text{ is the minimum} \\ Sp(i, j-1), & \text{if } D(i, j-1) \text{ is the minimum} \end{cases}$

$Sp(i, 0) = i; Sp(0, j) = 0; where 1<i<N, 1<j<M.$
Implementation on FPGA

- Four Loops to Guarantee Streaming
- Two-Phase Precision Reduction
- Support for Multi FPGAs
Implementation on FPGA

- Normalizer

Tuple $\Delta_1 \rightarrow$ shifter $\rightarrow$ Tuple $2M+1$, shifter $\rightarrow$ Tuple $2M+K+1$, Tuple $= \frac{\text{Tuple} - \text{mean}}{\text{std}}$

- Hybrid lower bound

Tuple in $\rightarrow$ LB_pDTW, envelope, distance $\rightarrow$ Reversed LB_Keogh, distance $\rightarrow$ LB_Keogh, distance $\rightarrow$ Max, distance $\rightarrow$ +, lower bound distance
Implementation on FPGA

\[ D(i, j) = \text{dist}(s_i, p_j) + \min\{D(i, j-1), D(i-1, j-1)\} \]

<table>
<thead>
<tr>
<th>PE</th>
<th>PE1</th>
<th>PE2</th>
<th>PE3</th>
<th>PE4</th>
<th>PE5</th>
<th>PE6</th>
<th>PE7</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>8</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>7</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>time</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>P2=5</td>
<td>INF</td>
<td>11(1)</td>
<td>5(2)</td>
<td>2(2)</td>
<td>6(2)</td>
<td>8(2)</td>
<td>11(5)</td>
</tr>
<tr>
<td>P1=0</td>
<td>INF</td>
<td>8(1)</td>
<td>1(2)</td>
<td>4(3)</td>
<td>9(4)</td>
<td>7(5)</td>
<td>9(6)</td>
</tr>
<tr>
<td>P7=0 INF</td>
<td>26(1)</td>
<td>19(1)</td>
<td>23(2)</td>
<td>16(2)</td>
<td>12(2)</td>
<td>14(2)</td>
<td>17(11)</td>
</tr>
</tbody>
</table>

DTW distance

Subsequence FIFO

Tuple router

Result router

Pattern RAM

FIFO Tuple router Result

Pattern valid

P out

valid

tuple

busy

value 8 1 4 9 7 9 6 0 8 9 6 7 7 3

time 1 2 3 4 5 6 7 8 9 10 11 12 13 14

PE PE1 PE2 PE3 PE4 PE5 PE6 PE7 PE1 PE2 PE3 PE4 PE5 PE6 PE7

P2=5 INF 11(1) 5(2) 2(2) 6(2) 8(2) 11(5) 7(7) 5(8) 3(8) 7(8) 8(8) 10(11) 11(14)
P1=0 INF 8(1) 1(2) 4(3) 9(4) 7(5) 9(6) 6(7) 0(8) 8(9) 9(10) 6(11) 7(12) 7(13) 3(14)
Experimental Setup

- **CPU**: Intel i7-930+ 16G RAM + Window 7
- **FPGA**: Altera Stratix4s530
  - Combinational ALUTs: 362,568/424,960 (85%)
  - Dedicated logic registers: 230,160/424,960 (54%)
  - Memory bits: 1,902,512/21,233,664 (9%)
  - Fmax: 167.8MHz
- **X=10, Y=502, PE number W=512**
Experimental Results

- Dataset1: medical Data
  - This dataset has about 8G points, and we need to find a pattern of length 421 with $R = 5\%$

### Table 1: Time taken to search one year of ECG data

<table>
<thead>
<tr>
<th></th>
<th>UCR_DTW[1]</th>
<th>Our work</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECG</td>
<td>18.0 minutes</td>
<td>56 seconds</td>
<td>19.28</td>
</tr>
</tbody>
</table>
Dataset 2: speech recognition

- We download the CMU_ARCTIC speech synthesis databases, and construct a speech of 1 minute (1 million points) by splicing together the first 21 utterances of all the 1132 utterances.
Experimental Results

**FPGA and GPU:**

<table>
<thead>
<tr>
<th>EPG data set</th>
<th>D. Sart [2]</th>
<th>Our work</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>80.39s</td>
<td></td>
<td>7398</td>
</tr>
<tr>
<td>FPGA</td>
<td>2.24s</td>
<td>0.011s</td>
<td>203</td>
</tr>
</tbody>
</table>

- To GPU & software: we use computation-reuse technique to exploit the coarse-grained parallelism, and the fine-grained parallelism can be only exploited by FPGA.
- To FPGA: we use both lower bound technique and computation-reuse technique.
Conclusions and Future Work

Conclusions

- IOT systems propose a lot of time series data
- Sensors and computing clusters (the cloud) have different requirements on tasks, so the problem is how to design a proper data manage system in order to help people to use these data
- For similarity search, which is a basic task for understanding and analyzing the streaming time series data, we proposed an FPGA acceleration architecture.

Future Work

- Explore the System Architecture for timing series data analysis to support the IOT data management system
  - Find the system arch patterns, and design the AS-system.
Future Work

No history data involved

- Similarity Search
- Segmentation
- Correlation Discovery
- Motif Discovery
- Visualization
- Classification
- Clustering
- Data Privacy
- Prediction
- Burst Detection
- Novelty/Anomaly detection
- Rule Discovery

May have real time req

History data analyses

Future Work
Reference


Thank you!

For other domain specific accelerations, such as graph theoretic algorithms, sparse matrix decomposition, search apps, video apps. Please refer to my webpage:

http://nics.ee.tsinghua.edu.cn/people/wangyu/
Domain Specific Accelerators: Opportunities for Software Library Replacement

John D. Davis
Researcher
Microsoft Research Silicon Valley
In Collaboration with Chuck Thacker, Eric Chung, Srinidhi Kestur, Lintao Zhang, Fang Yu, Zhangxi Tan, & Ollie Williams
The Virtuous Cycle is Broken!

- Doubling of transistors every 18-24 months
- End of Dennard Scaling

- Innovative Applications
- Miniaturization
- Lowered Costs
- Compute Capability & Efficiency
Conventional Wisdom, Circa 2000

Microprocessor Trends

Clock Frequency (MHz)

15 GHz Processor (100 Watts)

“The Multicore Revolution”

Source: http://cpudb.stanford.edu
Conventional Wisdom, Circa 2005

Multicore Trends

The Power Wall

- 16 cores @ 3.6GHz (100W)

Core Count x Frequency (n x GHz)
The Capability Wall

Capabilities (Battery Life, Performance, Killer Apps)

No Longer Can Rely on General Purpose Hardware
Improvements to Enable More Capabilities
Breaking Through the Capability Wall

10-100X Gap in Efficiency Between General Purpose Processors and Dedicated Hardware

*Source: Ning Zhang and Bob Brodersen, ISSCC data
Outline

- Motivation
- HW Accelerators & Goals
- Parallel SAT Solver
- Matrix-Vector Multiplication Engine
- Conclusions
HW Accelerators & Goals

- HW accelerators
  - Applications (PSAT)
  - Libraries (MVM)
  - Language

Common computation architectures to broaden accelerator utility.

- Customized memory architecture
- Compressed data representations
- Precision vs. energy efficiency
Exploring the Role of FPGAs

- Transistors are abundant, power is scarce
  - Utilize abundant silicon for FPGA fabrics
  - Energy efficient and post-silicon flexibility

Challenges

- FPGAs incur large reconfiguration overheads
- Must provide significant advantages over other architectures (many-core, GPGPU)
- What are the right applications?

Exploration enabled by the BEE3
We built it!

Vehicle for research in computer system architecture

- “BEE3”: Berkeley Emulation Engine, version 3
- 4 FPGAs (3 types of FPGAs)
  - Logic-focused, DSP-focused or Embedded Processor-focused
- 64 GB DDR2 DRAM
  - 2 DRAM channels per FPGA, 2 DIMMs per channel
- FPGA Ring Interconnect
- Plenty of I/O to connect to the BEE3
  - 10 GbE, 1 GbE, PCI-Express, QSH
**HW/SW Co-design Strategy**

- Two design styles
  - Directly translate SW → HW (generally FSMs)
    - Easy to debug and compare to SW system
  - Composable building blocks
    - Leverage domain (App + HW) expertise
    - Target FPGA hard macros

- General requirements for FPGAs
  - No reconfiguration
  - Generalized solution → library-like functionality
Parallel SAT Solver
Accelerating SAT Solving

- Determine whether a given boolean formula can be true
  \[(X_1 \lor X_5 \lor \overline{X}_9) \land (\overline{X}_8 \lor X_2 \lor \overline{X}_1)\]

- 3SAT is the first known NP-complete problem
  - Often used to prove that other problems are NP-complete

Applications of SAT:
- Formal verification of circuit design
- Cryptography attacks
- Solve other NP-complete problems

- SAT solver can take a long time
  - Some times hours, days, or even weeks
Hybrid HW/SW Solver

Loop{
  Branch decision
  -- set a variable

  Conflict Analysis
  -- backtrack, or finish

  Deduce
  -- loop through all related clauses, obtain inferred variables;

  90% time
  > 1000 CPU cycle/Inference

  10% time
}

FSB/HT/PCIe

CPU

FPGA

Software Solver

$X_1 \vee X_5 \vee \overline{X}_9$

$X_1=0$ (decision), $X_5=0$

$\Rightarrow X_9=0$
New Approach?

- Previous work
  - Map clauses to logic directly
  - Hours of reconfiguration time

- This is no longer only a logic problem!
  - It’s an architecture problem!
    - Design for FPGA fabric
    - Push computation close to storage (state)

- Transform logic problem into a memory indexing problem
FPGA Architecture

1: CPU communication module
2: Implication queue
3: Parallel inference engines
4: Inference multiplexer
5: Conflict inference detection
PSAT Summary: A HW/SW Solution

- An application-specific architecture
  - Reprogram memories for new instance
- Avoid global signal wires and careful pipelining
- Support tens of thousands of variables and clauses per FPGA
- Learned clause support
- BCP 5~16 times faster than the conventional software based approach
Dense & Sparse Matrix-Vector Multiplication
Matrix-Vector Multiply on FPGA
Matrix-Vector-Multiply is Critical HPC Kernel
- 10s of papers published/year on this topic

Existing works on GPU/CPU/FPGA
- Performance sensitive to matrix sparsity and formats
- Processor-centric data formats
- High power consumption (GPU/CPU)

FPGA opportunities
- Exploit custom variable-length formats
- Low power, large memory configurations
- Efficient, robust resource utilization
Objective: One Bitstream
To Rule Them All

- Build single FPGA bitfile library for $\vec{y} = A\vec{x}$
- Handle large-scale inputs ($\geq$ GB)
- Avoid costly run-time reconfiguration
- Exploit bit-level manipulation
- Dense and sparse inputs
- Process multiple sparse matrix formats
  - COO, CSR, Dense, DIA, ELL, etc.
Matrix-Vector Multiply (Dense)

\[
\begin{pmatrix}
A_{00} & A_{01} & A_{02} & A_{03} \\
A_{10} & A_{11} & A_{12} & A_{13} \\
A_{20} & A_{21} & A_{22} & A_{23} \\
A_{30} & A_{31} & A_{32} & A_{33} \\
A_{40} & A_{41} & A_{42} & A_{43}
\end{pmatrix}
\times
\begin{pmatrix}
x_0 \\
x_1 \\
x_2 \\
x_3
\end{pmatrix}
=
\begin{pmatrix}
y_0 \\
y_1 \\
y_2 \\
y_3 \\
y_4
\end{pmatrix}
\]

\[\vec{y} = A \vec{x}\]
Universal MVM (Dense Mode)

Universal Format Decoder

Matrix Memory (A)

Gaxpy Control

Tiled DMA Engine

Rows

Gaxpy PIPE 0

Gaxpy PIPE 1

Gaxpy PIPE 2

Gaxpy PIPE 3

Vector Memory (x)

Vector Memory (y)
Matrix-Vector Multiply (Sparse)

\[
\begin{bmatrix}
    A_{00} & A_{03} \\
    A_{12} & \\
    A_{21} & A_{32} \\
    A_{41} & A_{43}
\end{bmatrix} \times \begin{bmatrix}
    x_0 \\
    x_1 \\
    x_2 \\
    x_3
\end{bmatrix} = \begin{bmatrix}
    y_0 \\
    y_1 \\
    y_2 \\
    y_3
\end{bmatrix}
\]

\[\vec{y} = A \vec{x}\]
## Sparse Matrix Formats

### Data Array

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>

```
1 4 3 7 2 9 5 8 7
```
Coordinate (COO) Format

Data Array

\[
\begin{array}{cccccccccc}
1 & 4 & 3 & 7 & 2 & 9 & 5 & 8 & 7 \\
\end{array}
\]

Row Index

\[
\begin{array}{cccccccccc}
0 & 0 & 1 & 1 & 2 & 2 & 3 & 3 & 3 \\
\end{array}
\]

Column Index

\[
\begin{array}{cccccccccc}
0 & 2 & 0 & 1 & 2 & 3 & 0 & 1 & 3 \\
\end{array}
\]

COO Overhead = \( \text{Nonzeros} \times (4B + 4B) \)
Compressed Sparse Row (CSR) Format

Data Array

Row Pointer

Column Index

CSR Overhead = \( \text{Nonzeros} \times 4B + \text{Rows} \times 4B \)
ELLPACK (ELL) Format

Data w/ Padding

<table>
<thead>
<tr>
<th>1</th>
<th>4</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>7</td>
<td>*</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>*</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>7</td>
</tr>
</tbody>
</table>

Column Metadata

<table>
<thead>
<tr>
<th>0</th>
<th>2</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>*</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>*</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

\[ \text{ELL Overhead} = 4B \times \text{Rows} \times k + \text{DataPad} \times 8B \]
What’s wrong with these formats?

“I want you to find a bold and innovative way to do everything exactly the same way it’s been done for 25 years.”
FPGA Bit-Vector Format

Data Array

Bit Vector (BV)

BV Overhead = \textit{Rows} \times \textit{Cols} \times 1\text{bit}
Compressed Bit Vector (CBV)

Data Array

1 4 3 7 2 9 5 8 7

Compressed Bit Vector (CBV)

1 0 1 0 1 1 0 0 0 0 0 1 1 1 1 1 0 1

32-bit “zero” fields

CBV Overhead = Nonzeros × 1bit + ZeroClusters × 32bit
Compressed Variable BV (CVBV)

Data Array

1 4 3 7 2 9 5 8 7

Compressed Variable BV (CVBV)

1 0 1 0 1 1 0 0 0 0 0 1 1 1 1 1 0 1

1 (0,1) 1 (0,1) 1 (0,4) 1 1 1 1 (0,1) 1

4-bit header + \{4,8,...,32\}-bit zero field

CVBV Overhead \~ input-dependent
Sparse Format Comparison

CVBV storage (CSR-normalized) across 2547 matrices

Matrix column size

CVBV/CSR
Universal Matrix Format Decoder

Universal Format/ CVBV Decoder

A data streams

Gaxpy Control

Rows

Gaxpy PIPE 0
private cache (x)

Gaxpy PIPE 1
private cache (x)

Gaxpy PIPE 2
private cache (x)

Gaxpy PIPE 3
private cache (x)

Vector Memory (y)
Specify matrix format descriptors
  - Fixed/variable length, padding, index/ptr, etc.
Translates row/column into sequence #
Generate *BV (reduce storage/BW)
Generate modified COO (consumed by PEs)
  - Row index, nonzero count per row, col indices
**Algorithm 1** Universal Matrix Format Decoder.

**Input:** queue *streams*[3]

**Output:** Compressed data, rows, columns

1: data = streams[0].head
2: cx = streams[1].head
3: rx = streams[2].head

5: for i = 0 → NNZ − 1 do
6:   r = RowAddress? rx : ((rowStream.idx-1)/K)
7:   c = cx + ((pivot == -1) ? r : pivot)
8:   stream[0].dequeue()
9:   if stream[1] then
10:      stream[1].dequeue()
11:   end if
12:   if RowAddress then
13:      stream[2].dequeue()
14:   else if (rx − streams[1].idx) > 1 then
15:      stream[2].dequeue()
16:   end if
17: end for
## Performance/Area Results

<table>
<thead>
<tr>
<th></th>
<th>PEs</th>
<th>LUT (% area)</th>
<th>RAM (% area)</th>
<th>DSP (% area)</th>
<th>GFLOPs (Peak)</th>
<th>GFLOPs (off-chip)</th>
<th>BW (% peak)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dense V5-LX155T</td>
<td>16</td>
<td>72%</td>
<td>86%</td>
<td>88%</td>
<td>3.1</td>
<td>0.92</td>
<td>64.7</td>
</tr>
<tr>
<td>Dense V6-LX240T</td>
<td>32</td>
<td>71%</td>
<td>63%</td>
<td>56%</td>
<td>6.4</td>
<td>1.14</td>
<td>80</td>
</tr>
<tr>
<td>Dense+Sparse V5</td>
<td>16</td>
<td>74%</td>
<td>87%</td>
<td>91%</td>
<td>3.1</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sparse Inputs</th>
<th>V5-LX155T (Ours)</th>
<th>HC-1 (32 PE)¹</th>
<th>Tesla S1070²</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GFLOPS / BWUsed</td>
<td>GFLOPS / BWUsed</td>
<td>GFLOPS / BWUsed</td>
</tr>
<tr>
<td>dw8192</td>
<td>0.10 / <strong>10.3%</strong></td>
<td>1.7 / <strong>13.2%</strong></td>
<td>0.5 / <strong>3.1%</strong></td>
</tr>
<tr>
<td>t2d_q9</td>
<td>0.15 / <strong>14.4%</strong></td>
<td>2.5 / <strong>19.3%</strong></td>
<td>0.9 / <strong>5.7%</strong></td>
</tr>
<tr>
<td>epb1</td>
<td>0.17 / <strong>17.1%</strong></td>
<td>2.6 / <strong>20.2%</strong></td>
<td>0.8 / <strong>4.9%</strong></td>
</tr>
<tr>
<td>raefsky1</td>
<td>0.20 / <strong>18.5%</strong></td>
<td>3.9 / <strong>29.0%</strong></td>
<td>2.6 / <strong>15.3%</strong></td>
</tr>
<tr>
<td>psmigr_2</td>
<td>0.20 / <strong>18.6%</strong></td>
<td>3.9 / <strong>29.6%</strong></td>
<td>2.8 / <strong>16.7%</strong></td>
</tr>
<tr>
<td>torso2</td>
<td>0.04 / <strong>4.0%</strong></td>
<td>1.2 / <strong>9.1%</strong></td>
<td>3.0 / <strong>18.3%</strong></td>
</tr>
</tbody>
</table>

¹ [Nagar et al., A Sparse Matrix Personality for the Convey HC-1, FCCM'11]
² [Bell et al., Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors, SC’09]
MVM Summary

- We defined CVBV/CBV sparse format
  - 25% reduction in storage/bandwidth compared to well-known CSR
  - Exploits bit-level manipulation of FPGA
- Single bit file for dense AND sparse MVM
  - Universal matrix format decoder
  - DMA and caches for memory management
  - Stall-free accumulator
  - Scalable design, implemented on multiple platforms
Conclusions

- Demonstrated HW as SW library replacement
  - Bottom up approach (Time consuming/not scalable)
- Pros:
  - Common computation architecture and input insensitive
  - Customized memory architecture
  - Compressed data representations
  - Other energy efficiency tools to exploit
- Cons:
  - Time consuming: requires designers
Future Directions

- Moving beyond manycore and GPUs
- Need tools to automate HW/SW co-design
  - Granularity?
  - Algorithm specification?
- HW building blocks?
  - IP Integration issues
- Future FPGA Architecture?
  - More custom building blocks?
- Software/OS support
  - Fast communication and synchronization
  - Accelerators as 1st class building blocks
Questions?
Back up
Results without Learning

- BCP 6.7 – 38.6 times faster than the conventional software based approach

![Bar chart showing CPU cycles for 3-SAT, 4-SAT, 5-SAT, and 6-SAT for different hardware configurations: CPU, FPGA (HT), and FPGA (PCIe). BCP 6.7 is significantly faster than the conventional software based approach.](chart)
Results with Learning

- BCP 5~16 times faster than the conventional software based approach