Automatic High-Performance Kernel Generation for DL Accelerators: Specialization to Generalization*

Prasanth Chatarasi

Research Staff Member @ AI Hardware Group, IBM T.J. Watson Research Center, YorkTown Heights, NY, USA
prasanth@ibm.com

Google Brain, May 26th, 2021

*Work done during PhD studies in Habanero Research Group at Georgia Institute of Technology
Deep Learning (DL) Accelerators

- Emerged to address the demands of DL models training and inferences
  - A large array of processing elements to provide high performance
  - Direct communication between PEs for energy efficiency
    - PE & PE requires ~3x less energy compared to PE & L2
Design Architecture of Deep Learning (DL) Compilers

Key steps in the flow:

• **Graph compiler** performing graph-level optimizations
  - e.g., Node/Layer fusion

• **Kernel compiler** performing kernel-level optimizations
  - e.g., loop scheduling

Fig. 2. The overview of commonly adopted design architecture of DL compilers.

Kernel Compilers of DL Compiler stack

Three major approaches for kernel compilers:

- Stitch manually-written kernel libraries for each node of the optimized graph
- Automatically generate entire kernels corresponding to each node of the graph
- Hybrid approach combining some parts with manually-written kernels and other parts with automatic generation

Fig. 4. Overview of hardware-specific optimizations applied in DL compilers.

Manually-written vs Automatically-generated

1. Manually-writing kernels is not a scalable approach, but a good temporary solution!
   • Kernels are evolving rapidly, for, e.g., many variants of convolutions with different shapes/sizes
   • ML Accelerators also evolving so quickly, for, e.g., TPU V1, V2, and V3 with different capabilities
   • Need automatic kernel generation along with mappers and cost models that does fine-grained reasoning of both kernels and hardware to achieve peak performance!

2. Automatic kernel generation is a scalable solution, but several challenges.
   • Need “GOOD” hardware abstractions to capture various accelerators
   • Need “GOOD” mapping abstractions to capture various mapping strategies of workloads
   • Need “GOOD” cost models to estimate performance and drive mappers/auto-tuners
   • Several variants:
     • Fixed hardware + Fixed kernel,
     • Fixed hardware + Allow different kernel possibilities
     • Allow various hardware choices + Allow different kernel possibilities
Overview of today’s talk

1. Introduction & Background

2. Vyasa: A High-performance Vectorizing Compiler for Tensor Operations onto Xilinx AI Engine (2D SIMD unit)
   - Fixed hardware + Allow different kernel possibilities

3. PolyEDDO: A Polyhedral-based Compiler for Explicit De-coupled Data Orchestration (EDDO) architectures
   - Allow various hardware choices + Allow different kernel possibilities

4. Conclusions
Overview of today’s talk

1. Introduction & Background

2. Vyasa: A High-performance Vectorizing Compiler for Tensor Operations onto Xilinx AI Engine (2D SIMD unit)
   • Fix hardware + Allow different kernel possibilities

3. PolyEDDO: A Polyhedral-based Compiler for Explicit De-coupled Data Orchestration (EDDO) architectures
   • Allow various hardware choices + Allow different kernel possibilities

4. Conclusions

"Vyasa: A High-Performance Vectorizing Compiler for Tensor Convolutions on the Xilinx AI Engine"
Prasanth Chatarasi, Stephen Neuendorffer, Samuel Bayliss, Kees Vissers, and Vivek Sarkar
Proceedings of the 24th IEEE High Performance Extreme Computing Conference (HPEC’20)
Xilinx Versal AI Engine

• A High Performance & Power Efficient VLIW SIMD Core
  • Part of Xilinx Versal Advanced Compute Acceleration Platform
    • Scalar Engines (CPUs), Adaptable Engines (Programmable logic), Intelligent Engines (AI Engines)

AI Engine: Terminology
Key architectural features of AI Engine

1) **2D SIMD datapath for fixed point**
   - Reduction within a row/lane
   - #Columns depend on operand precision
     - 32-bit types: 8 rows x 1 col
     - 16-bit types: 8 rows x 4 col (or) 16 rows x 2 col
     - 8-bit types: 16 rows x 8 col

2) **Shuffle Interconnection network**
   - Between SIMD and vector register file
   - Supports arbitrary selection of elements from a vector register
     - Some constraints for 16-/8-bit types
   - Selection parameters are provided via vector intrinsics
Problem Statement & Challenges

**Problem statement:** How to implement high-performance primitives for tensor convolutions on AI Engine?

- **Current practice:** Programmers manually use vector intrinsics to program 2D SIMD unit and also explicitly specify shuffle network parameters for data selection.

- **Challenges:** Error prone, written code may not be portable to a different schedule or data-layouts, daunting to explore all choices to find best implementation, tensor convolutions vary in sizes and types.

**Our approach:** Vyasa, a domain-specific compiler to generate high performance primitives for tensor convolutions from a high-level specification.

Vyasa means “compiler” in the Sanskrit language, and also refers to the sage who first compiled the Mahabharata.
Our high-level approach (Vyasa)

In this talk, I focus on Step-3 and Step-4 leveraging Shuffle Network and 2D SIMD datapath!
Running Example — CONV1D

A sample schedule: Unroll w-loop and Vectorize x-loop (VLEN: 16)

\[
\begin{align*}
O(0:15) &= W(0) \times I(0:15) \\
O(0:15) &= W(1) \times I(1:16) \\
O(0:15) &= W(2) \times I(2:17) \\
O(0:15) &= W(3) \times I(3:18)
\end{align*}
\]

for (x = 0; x < 16; x++)
for (w = 0; w < 4; w++)
O[x] += I[x+w]*W[w];
Challenges

O(0:15) += W(0) * I(0:15)
O(0:15) += W(1) * I(1:16)
O(0:15) += W(2) * I(2:17)
O(0:15) += W(3) * I(3:18)

V1 = VLOAD(I, 0:15);
V2 = BROADCAST(W, 0);
V3 = VMAC(V1, V2);
V4 = VLOAD(I, 1:16);
V5 = BROADCAST(W, 1);
V6 = VMAC(V3, V4, V5);
V7 = BROADCAST(W, 2);
V8 = VMAC(V3, V6, V7);
V9 = BROADCAST(W, 3);
V3 = VMAC(V3, V8, V9);
VSTORE(O, 0:15, V3);

No support for unaligned loads
O(0:15) += W(0) * I(0:15)
O(0:15) += W(1) * I(1:16)
O(0:15) += W(2) * I(2:17)
O(0:15) += W(3) * I(3:18)

No support for broadcast operations
V1 = VLOAD(I, 0:15);
V2 = BROADCAST(W, 0);
V3 = VMAC(V1, V2);
V4 = VLOAD(I, 1:16);
V5 = BROADCAST(W, 1);
V6 = VMAC(V3, V4, V5);
V7 = BROADCAST(W, 2);
V8 = VMAC(V3, V6, V7);
V9 = BROADCAST(W, 3);
V3 = VMAC(V3, V8, V9);
VSTORE(O, 0:15, V3);

V6 and V8 have 15 elements in common.
How to reuse them without loading again?

How to exploit multiple columns of 2D vector substrate?
1) Exploiting Vector Register Reuse

- Build “temporal reuse graph” with nodes being vector loads
  - Edge exists b/w nodes if there is at least one element in common

- AI Engine allows to create logical vector registers of length up to 1024 bits
  - Identify (aligned) connected components and assign each component to a vector register that can subsume the individual vector loads of the component.
  - Use shuffle interconnection network to select desired elements

Mathematical equations:

\[
\begin{align*}
O(0:15) &= W(0) \times I(0:15) \\
O(0:15) &= W(1) \times I(1:16) \\
O(0:15) &= W(2) \times I(2:17) \\
O(0:15) &= W(3) \times I(3:18)
\end{align*}
\]
2) Exploiting Spatial Locality

• Build “spatial locality graph” with nodes being scalar loads
  • Edge exists b/w nodes if they can be part of an aligned vector load
• Identify connected components
• AI Engine allows to create logical vector registers of length up to 1024 bits
  • Assign each connected component (aligned) to a logical vector register
  • Use shuffle interconnection network to select desired elements

\[
\begin{align*}
O(0:15) &+ W(0) \times I(0:15) \\
O(0:15) &+ W(1) \times I(1:16) \\
O(0:15) &+ W(2) \times I(2:17) \\
O(0:15) &+ W(3) \times I(3:18)
\end{align*}
\]
3) Grouping 1D Vector Operations

\[ O(0:15) += W(0) \times I(0:15) \]
\[ O(0:15) += W(1) \times I(1:16) \]
\[ O(0:15) += W(2) \times I(2:17) \]
\[ O(0:15) += W(3) \times I(3:18) \]

All the 4 operations are performed with a single load of V1 and V2 (maximum reuse)
Our high-level approach (Vyasa)

Auto-tuner explores the space of schedules related to loop and data-layouts.

Loop transformations:
1. Choice of vectorization loop
2. Loop reordering
3. Loop unroll and jam

Data-layout choices:
1. Data permutation
2. Data tiling (blocking)

We assume that workload memory footprint fits into a AI Engine local scratchpad memory (128KB)
Evaluation

- **CONV2D workloads (only for this talk)**
  - CONV2D in Image processing pipelines/Computer Vision (CV)
    
    HALIDE CODE: \( O(x, y) += W(r, s) * I(x+r, y+s); \)
  - CONV2D in Convolutional Neural Networks (CNNs)
    
    HALIDE CODE: \( O(x, y, k, n) += W(r, s, c, k) * I(x+r, y+s, c, n); \)

- **Comparison variants**
  - Roofline peak
    - 32-bit types: 8 MACs/cycle
    - 16-bit types: 32 MACs/cycle
  - Expert-written and tuned kernels for Computer Vision

### AI Engine configurations

<table>
<thead>
<tr>
<th>Parameter</th>
<th>32-bit</th>
<th>16-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D SIMD data path</td>
<td>8 x 1</td>
<td>16 x 2</td>
</tr>
<tr>
<td>Peak compute</td>
<td>8 MACs/cycle</td>
<td>32 MACs/cycle</td>
</tr>
<tr>
<td>Scratchpad memory</td>
<td>128 KB @ 96B/cycle</td>
<td></td>
</tr>
<tr>
<td>Scratchpad memory ports</td>
<td>32B 2 read and 1 write</td>
<td></td>
</tr>
<tr>
<td>Vector register file</td>
<td>256 B</td>
<td></td>
</tr>
</tbody>
</table>
• **Expert-written codes** are available only for 3x3 and 5x5 filters
  • Available as part of the Xilinx’s AI Engine compiler infrastructure

• **Auto-tuner was able to find better schedules**
  • Especially non-trivial unroll and jam factors
Evaluation: CONV2D’s in CV (256x16) with various Filters

<table>
<thead>
<tr>
<th>Filter Size</th>
<th>MACs/Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>2x2</td>
<td>21.59</td>
</tr>
<tr>
<td>3x3</td>
<td>21.76</td>
</tr>
<tr>
<td>4x4</td>
<td>28.53</td>
</tr>
<tr>
<td>5x5</td>
<td>23.65</td>
</tr>
<tr>
<td>6x6</td>
<td>27.54</td>
</tr>
<tr>
<td>7x7</td>
<td>26.20</td>
</tr>
<tr>
<td>8x8</td>
<td>29.33</td>
</tr>
<tr>
<td>9x10</td>
<td>26.49</td>
</tr>
<tr>
<td>10x10</td>
<td>28.45</td>
</tr>
<tr>
<td>11x11</td>
<td>27.08</td>
</tr>
<tr>
<td>Geo. Mean</td>
<td>25.92</td>
</tr>
</tbody>
</table>

- **Even-sized filters (except 2x2), our approach achieved close to peak**
  - 87% for 16-bit and 95% for 32-bit

- **Odd-sized filters, our approach padded each row with an additional column**
  - For 16-bit type, number of reductions should be multiple of two (2 columns)
Evaluation: CONV2D’s in CNN’s (128x2x16)

**HALIDE CODE for REG CONV2D:** $O(x, y, k, n) += W(r, s, c, k) * I(x+r, y+s, c, n)$;

- **REG-CONV2D** (3x3, 5x5, 7x7)
  - Vectorization along Output width and Reduction along Filter channels
- **PW-CONV2D** (1x1), **SS-CONV2D** (1x3, 3x1), **FC-CONV2D** (1x1)
  - Vectorization along Output channels and Reduction along Filter channels
- **DS-CONV2D** (3x3) — *Padded each row*
  - Vectorization along Output width and Reduction along Filter width
Non-trivial data-layout choices

• 16-bit REG-CONV2D (3x3)
  • Vectorization along Output width and Reduction along Filter channels
  • For the fused vector operation ($W1 \times I1 + W2 \times I2$)
    • Data for ($I1, I2$) should be in a single vector register for the operation
    • $I1(0)$ and $I2(0)$ should be adjacent for shuffle network constraints
  • $(C/2)Y’X’(2)$ refers to first laying out an input block of two channels followed by width, height, and remaining channels.
Summary and Questions

• **Summary**
  • Manually writing vector code for high-performant tensor convolutions achieving peak performance is extremely challenging!
  • **Automatic kernel generation can be the key!**
    • Proposed a convolution-specific IR for easier analysis and transformations
    • Our approach (Vyasa) can work for any convolution variant regardless of its variations and shapes/sizes.
    • Achieved close to the peak performance for a variety of tensor convolutions

• **Questions**
  • How about beyond tensor-style operations?
    • E.g., fused convolutions (depth-wise + point-wise), non-rectilinear iteration spaces (Symmetric GEMM)
  • How about beyond the AI Engine?
    • E.g., other accelerators like IBM Rapid, MIT Eyeriss, NVDLA…
Overview of today’s talk

1. Introduction & Background

2. Vyasa: A High-performance Vectorizing Compiler for Tensor Operations onto Xilinx AI Engine (2D SIMD unit)
   • Fixed hardware + Allow different kernel possibilities

3. PolyEDDO: A Polyhedral-based Compiler for Explicit De-coupled Data Orchestration (EDDO) architectures
   • Allow various hardware choices + Allow different kernel possibilities

4. Conclusions

“Hardware Abstractions for targeting EDDO Architectures with the Polyhedral Model”
Angshuman Parashar, Prasanth Chatarasi, and Po-An Tsai,
11th International Workshop on Polyhedral Compilation Techniques (IMPACT’21)
ICDO vs EDDO Architectures

**Implicit Coupled Data Orchestration (ICDO)**
e.g., CPUs, GPUs

**Explicit Decoupled Data Orchestration (EDDO)**
e.g., IBM Rapid AI, NVIDIA Simba, NVDLA, Eyeriss, etc.

**EDDO architectures attempt to minimize data movement costs**
**Benefits**

- Dedicated (and often statically programmed) state machines more efficient than general cores
- Perfect “prefetching”
- **Buffet** storage idiom provides fine-grain synchronization and efficient storage, or scratchpads + Send/Recv synchronization
- Hardware mechanisms for reuse

*Explicit Decoupled Data Orchestration (EDDO)*
e.g., IBM Rapid AI, NVIDIA Simba, NVDLA, Eyeriss, etc.

---

Pellauer et. al., “Buffets: An Efficient and Composable Storage Idiom for Explicit Decoupled Data Orchestration”, ASPLOS 2019
EDDO Architectures

Challenges

1. **No single binary**: Collection of distinct binaries that program distributed state machines working together to execute algorithm
   - E.g., CNN layer on EDDO arch → ~250 distinct state machines.

2. **Reuse optimization** is critical for efficiency
   - E.g., CNN layer on EDDO arch → 480,000 mappings, 11x spread in energy efficiency, 1 optimal mapping
   - Need an optimizer or mapper

3. **Variety of EDDO architectures**, constantly evolving
   - Need an abstraction that Mapper and Code Generator will target

---

Explicit Decoupled Data Orchestration (**EDDO**) e.g., IBM Rapid AI, NVIDIA Simba, NVDLA, Eyeriss, etc.
Overall Compilation Flow

Mapper *(optimizer)*

PolyEDDO Code Generator

Arch-specific configuration generator

Configuration Binaries

Workload

Mapping

Arch-independent decoupled programs

EDDO Architecture described using Hardware Space-Time (HST)

Perf, Energy, Area

Analytical microarchitecture and energy model †

Data-movement activity counts


**Example 1 — Symbolic Hardware Space-Time (SHST)**

\[ \text{SpaceTime}_2 [s_2, t_2] \rightarrow \text{SpaceTime}_1 [s_1, t_1] : \]

- \( s_2 = 0 \) \& \( t_2 = 0 \) \&
- \( 0 \leq s_1 < 4 \) \& \( 0 \leq t_1 < 3 \)

Single L2, 4 L1s, 3 time-steps
- In each step, the L2 delivers a tile of data to each L1
- Across all these L1 time steps, the resident tile in L2 does not change. In effect, **time is stagnant for L2**
Example 2 — Symbolic Hardware Space-Time (SHST)

\[
\text{SpaceTime}_3[s_3, t_3] \rightarrow [\text{SpaceTime}_2[s_2, t_2] \rightarrow \text{SpaceTime}_1[s_1, t_1]] :
\]

\[
\begin{align*}
s_3 &= 0 & t_3 &= 0 \\
0 \leq s_2 &< 2 & 0 \leq t_2 &< 2 \\
0 \leq s_1 &< 4 & 0 \leq t_1 &< 3
\end{align*}
\]
Example 3 — Partitioned Buffers

\[ \Theta_{\text{HST}}(\text{DRAM}) = \text{SpaceTime}_3 [0, 0] \]
\[ \Theta_{\text{HST}}(\text{BufA}) = \text{SpaceTime}_3 [0, 0] \rightarrow \text{SpaceTime}_2 [s_2, t_2] \]
\[ \Theta_{\text{HST}}(\text{BufB}) = \text{SpaceTime}_3 [0, 0] \rightarrow \text{SpaceTime}_2 [s_2, t_2] \]
\[ \Theta_{\text{HST}}(\text{BufZ}) = \text{SpaceTime}_3 [0, 0] \rightarrow \text{SpaceTime}_2 [s_2, t_2] \]
\[ \Theta_{\text{HST}}(\text{OperandA}) = \text{SpaceTime}_3 [0, 0] \rightarrow \text{SpaceTime}_2 [s_2, t_2] \rightarrow \text{SpaceTime}_1 [s_1, t_1] \]
\[ \Theta_{\text{HST}}(\text{OperandB}) = \text{SpaceTime}_3 [0, 0] \rightarrow \text{SpaceTime}_2 [s_2, t_2] \rightarrow \text{SpaceTime}_1 [s_1, t_1] \]
\[ \Theta_{\text{HST}}(\text{Result}) = \text{SpaceTime}_3 [0, 0] \rightarrow \text{SpaceTime}_2 [s_2, t_2] \rightarrow \text{SpaceTime}_1 [s_1, t_1] \]

\[ \Theta_{\text{HST}}(\text{OperandA}) = \text{SpaceTime}_3 [0, 0] \rightarrow \text{SpaceTime}_2 [s_2, t_2] \rightarrow \text{SpaceTime}_1 [s_1, t_1] \]
\[ \Theta_{\text{HST}}(\text{OperandB}) = \text{SpaceTime}_3 [0, 0] \rightarrow \text{SpaceTime}_2 [s_2, t_2] \rightarrow \text{SpaceTime}_1 [s_1, t_1] \]

Workload mappings target SHST

<table>
<thead>
<tr>
<th>SHST</th>
<th>PHST</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{SpaceTime}_3 [s_3, t_3] \rightarrow \text{SpaceTime}_2 [s_2, t_2] \rightarrow \text{SpaceTime}_1 [s_1, t_1]</td>
<td>\rightarrow \text{DRAM} [s_3, t_3]</td>
</tr>
<tr>
<td>\rightarrow \text{BufA} [s_2, t_2]</td>
<td>\rightarrow \text{BufB} [s_2, t_2]</td>
</tr>
<tr>
<td>\rightarrow \text{BufZ} [s_2, t_2]</td>
<td>\rightarrow \text{OperandA} [2s_2 + s_1, t_2, t_1]</td>
</tr>
<tr>
<td>\rightarrow \text{OperandB} [2s_2 + s_1, t_2, t_1]</td>
<td>\rightarrow \text{Result} [2s_2 + s_1, t_2, t_1]</td>
</tr>
</tbody>
</table>
Example 4 — Eyeriss-like accelerator

SHST: $SpaceTime_4 [s_4, t_4] \rightarrow SpaceTime_3 [s_3, t_3] \rightarrow [SpaceTime_2 [s_2, t_2] \rightarrow SpaceTime_1 [s_1, t_1]]$

See paper for full HST

Observe how different the architecture is from CPUs and GPUs

Workload mappings target SHST
PolyEDDO Code Generator

Architecture HST
Workload
Mapping

T-relations
Decoupling

Tiling (T)-relations
Data Transfer (X)-relations

Reuse Analysis
Delta (Δ)-relations

Schedule creation
Δ schedules

AST generation

Decoupled ASTs
Mapping workloads (Tensor operations)

Workload Iteration Space

Tile

Tile

Project

Project

Project

SHST

L1

L2

L3

s1

s2

t1

t2
Mapping workloads (The Tiling-relation, T-relation)

Set of Tensor Coords

$\rightarrow$ MatrixA[m,k] : ...
MatrixB[k,n] : ...
MatrixZ[m,n] : ...

T-relation: Projection from SHST coordinate to a set of tensor coordinates
- Tells you what tiles of data must be present at that point in space-time to honor the mapping.
- Does not tell you how the data got there.
Decoupling — Breaking the hierarchy

Decouple

T-relations

HST

PHST

L3

DRAM

L2

BufA BufB BufZ

BufA BufB BufZ

BufA BufB BufZ

BufA BufB BufZ

L1

OperandA

OperandB

Result

MACCs


[MACC[s1, t1]] -> MulAcc[k, p, r] : …

Data transfer relations (X-relations)

Tensor coords

SHST

s1

s2

s3

t1

t2

t3

[166x189]B

[145x87]×

[196x87]×

[248x87]×

[97x59]OperandA

[129x43]OperandB

[393x189]Buf

[398x189]B

[345x189]Buf

[351x189]A

[436x196]BufZ

[319x86]×

[371x86]×

[423x86]×

[474x86]×

[909x480]BufB

[819x478]L2

[1021x584]DRAM

[946x585]L3

[858x480]BufA

[959x480]BufZ

[1151x479]BufB

[1100x479]BufA

[1201x479]BufZ

[793x151]L1

MACCs

[832x134]×

[887x134]×

[943x134]×

[998x134]×

[1074x134]×

[1129x134]×

[1185x134]×

[1240x134]×

[139x448]t1

[288x448]t1

[347x448]t1

[281x505]s1

[223x416]s1

[364x494]T-relations

[123x278]PHST

[217x376]HST

[268x293]Macros

[46x89]L1

[198x295]L3

[119x203]Buf

[124x189]B

[210x196]BufZ

[93x87]×

[145x87]×

[196x87]×

[248x87]×

[97x59]OperandA

[129x43]OperandB

[393x189]Buf

[398x189]B

[345x189]Buf

[351x189]A

[436x196]BufZ

[319x86]×

[371x86]×

[423x86]×

[474x86]×

[909x480]BufB

[819x478]L2

[1021x584]DRAM

[946x585]L3

[858x480]BufA

[959x480]BufZ

[1151x479]BufB

[1100x479]BufA

[1201x479]BufZ

[793x151]L1

MACCs

[832x134]×

[887x134]×

[943x134]×

[998x134]×

[1074x134]×

[1129x134]×

[1185x134]×

[1240x134]×

[139x448]t1

[288x448]t1

[347x448]t1

[281x505]s1

[43x457]L1

[89x590]L2

[157x519]s2

[282x573]t2

[59x683]L3

SHST

s1

s2

s3

t1

t2

t3

[711x20]36
REUSE ANALYSIS

Local Temporal Reuse
REUSE ANALYSIS

Fill from Peer
REUSE ANALYSIS

Fill from parent
REUSE ANALYSIS

Parent Multicast/Spatial Reduction
OPTIMIZATION PROBLEM (FOR A SINGLE MAPPING!)

Options:
1. Enumerate all possibilities and find optimum solution
2. Use a heuristic
3. Expose choices to mapping (and thereby the mapspace)
**POLYEDDO**

- **T-relation generation**
  - Architecture HST
  - Workload
  - Mapping

- **Decoupling**
  - Tiling (T)-relations

- **Reuse Analysis**
  - Data Transfer (X)-relations
  - Delta (Δ)-relations

- **Schedule creation**
  - Δ schedules

- **AST generation**
  - Decoupled ASTs

---

*Described in paper*
• Present capability: build generated code against an EDDO emulator (automatically configured from the PHST)
Summary and Questions?

• **Summary**
  - HST (Hardware Space-Time) – an abstraction for EDDO architectures represented using the Polyhedral Model
  - PolyEDDO (WIP) – an analysis and code-generation flow based on HST

• **Research questions**
  - How do we think about mapping imperfectly-nested loops to generic EDDO architectures?
  - How do we capture sparsity extensions of the accelerators?
• Ph.D. Advisors:
  • Vivek Sarkar (advisor), and Jun Shirako (co-advisor)

• Collaborators
  • Albert Cohen, Martin Kong, Tushar Krishna, Hyoukjun Kwon, John Mellor-Crummey, Karthik Murthy, Stephen Neuendorffer, Angshuman Parashar, Micheal Pellauer, Kees Vissers, and others

• Other mentors
  • Kesav Nori, Uday Bondhugula, Milind Chabbi, Shams Imam, Deepak Majeti, Rishi Surendran, and others

• IBM Research, Habanero & Synergy Research Group Members
Backup
Why do we need accelerators?

1) DNN models have tight constraints on latency, throughput, and energy consumption, esp. on edge devices

2) DNN models have trillions of computations
   Need high throughput — Makes CPUs inefficient

3) DNN models involve heavy data movement
   Need to reduce energy — Makes GPUs inefficient
Landscape of DNN Accelerators