ADVANCING COMPILER OPTIMIZATIONS FOR GENERAL-PURPOSE & DOMAIN-SPECIFIC PARALLEL ARCHITECTURES

Prasanth Chatarasi

PhD Thesis Defense

Habanero Extreme Scale Software Research Group
School of Computer Science
Georgia Institute of Technology

July 27th, 2020

Georgia Tech School of Computer Science

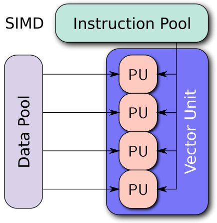
Disruption in Computer Hardware

- **Transistor scaling is reaching its limits (7nm today)**
  - Leading to the end of Moore's law

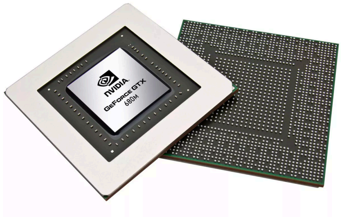
General-Purpose Parallel Architectures

Multi-core CPUs

Many-core CPUs

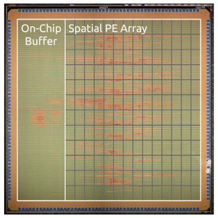


SIMD

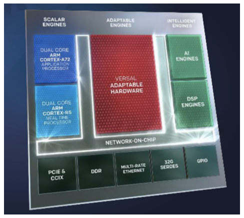


GPGPUs

Domain-specific Parallel Architectures

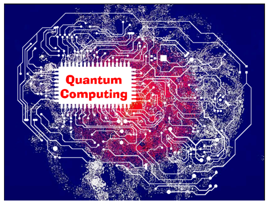


Spatial accelerators



Specialized SIMD

Thread Migratory



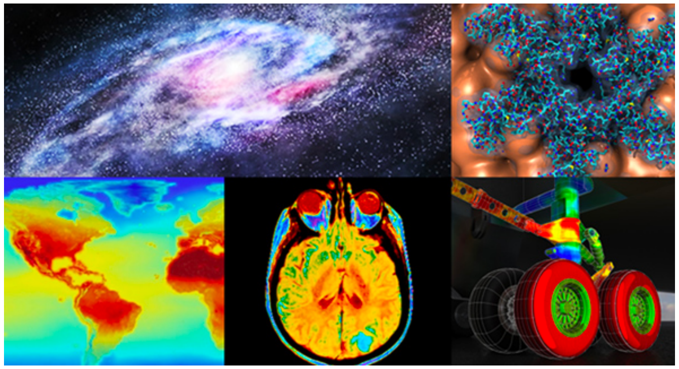
Quantum

These architectures are evolving rapidly!

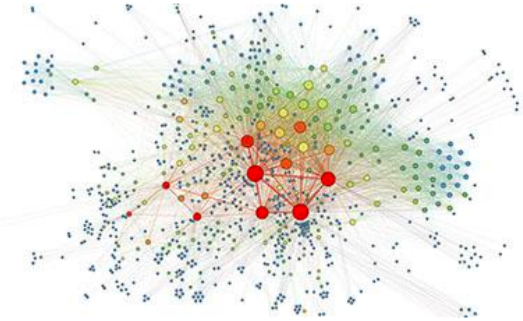
Images are taken from public domain

Application domains that demand high performance are also increasing

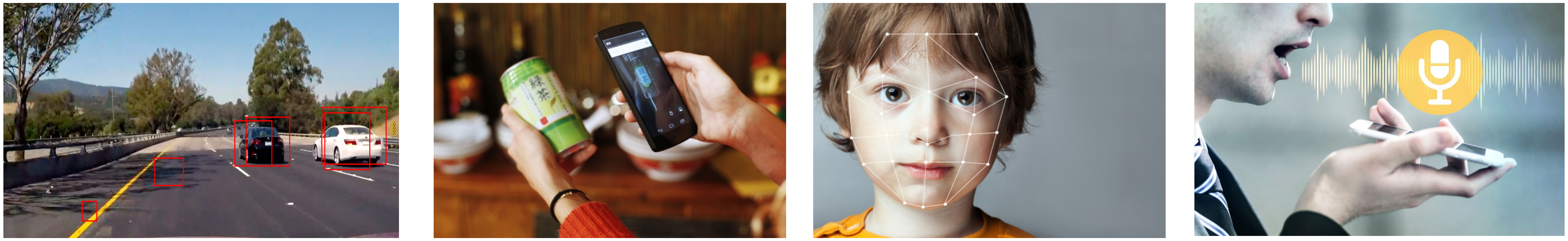
Scientific computing applications



Large scale graph processing



Machine learning (Deep Neural Networks)



Furthermore, these domains are rapidly evolving with new algorithms!

Ways to achieve high-performance

1) Ninja/Expert programmers

- Achieve close to peak performance
- Hard to port to new hardware platforms
- Only a small fraction of developers are Ninja programmers

2) High-performance libraries

cuDNN

- Easy to develop high performance applications
- Not portable across platforms
- Hard to support rapidly evolving applications
- Inhibits optimizations across library calls

3) Optimizing compilers

LLVM COMPILER INFRASTRUCTURE

MLIR

- Easy to develop high performance applications
- Portable across platforms
- Easily supports rapidly evolving applications
- Enables full-program optimizations
- **Promising direction, but requires advancements!**

Thesis statement

“Given the increasing demand for performance across multiple application domains and the major disruptions in future computer hardware as we approach the end of Moore’s Law, our thesis is that advances in compiler optimizations are critical for enabling a wide range of applications to exploit future advances in both general-purpose and domain-specific parallel architectures.”

Key Contributions

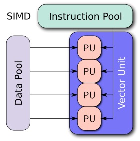
Advancing Compiler Optimizations for General-Purpose Parallel Architectures

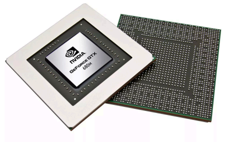
1) Analysis and optimization of explicitly parallel programs (PACT’15)

Multi-core/Many-core CPUs

2) Unification of storage transformations with loop transformations (LCPC’18)

Vector Units (SIMD, SIMT)





Advancing Compiler Optimizations for Domain-Specific Parallel Architectures

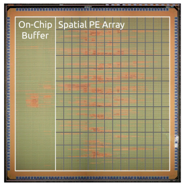
3) Domain-specific compiler for graph analytics on thread migratory hardware (MCHPC’18)

Thread migratory (EMU)

EmuTechnology

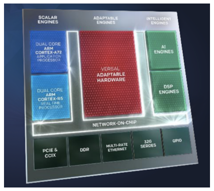
4) Data-centric compiler for DNN operators on flexible spatial accelerators (ArXiv’20)

Flexible Spatial accelerators



5) Domain-specific compiler for tensor convolutions on 2D SIMD units (Under submission)

Specialized vector units (AI Engine)



Analysis and Optimizations of Explicitly-Parallel Programs

"Polyhedral Optimizations of Explicitly Parallel Program"
Prasanth Chatarasi, Jun Shirako, and Vivek Sarkar,
In Proceedings of the 24th International Conference on Parallel Architecture and Compilation (PACT'15)
(One of four papers selected for Best Paper session)

Explicit parallel software on the rise!

- **Parallel programming of multi-cores, many-cores in CPUs, GPUs have become mainstream**
  - E.g., OpenMP for CPUs, CUDA for GPUs

- **Programmers explicitly specify parallelism in the program**

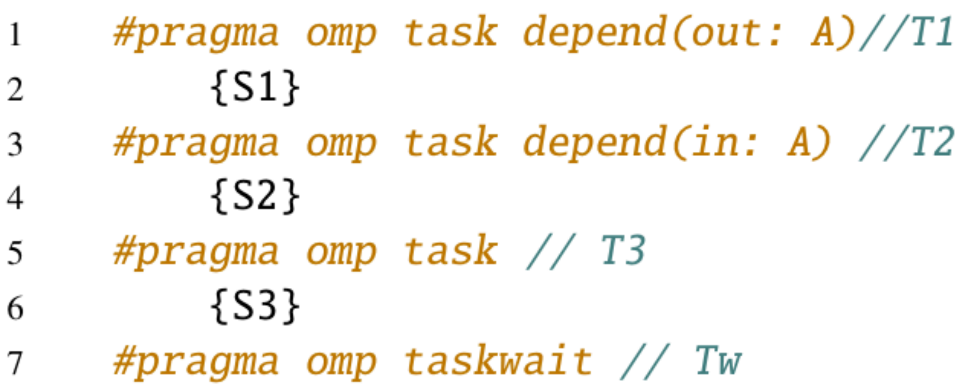
Key Challenges:

1) ***How to extend foundations of optimizing compilers to support explicit parallelism?***

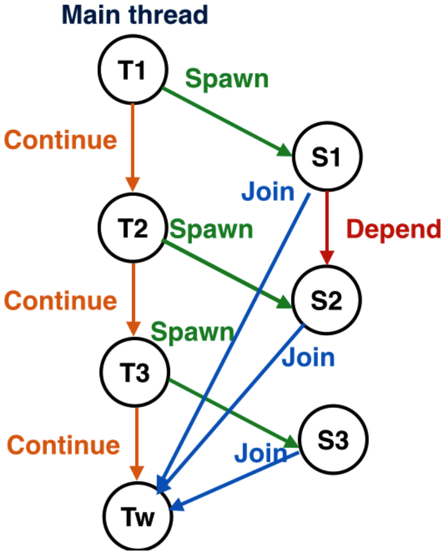
2) ***Can explicit-parallelism be used to refine conservative (imprecise) dependences?***

Background: Explicit Parallelism

- **Parallel programs have partial execution order**
  - Described by Happens-before relations
- **Loop-level parallelism (since OpenMP 1.0)**
  - Iterations of the loop can be run in parallel
- **Task-level parallelism (since OpenMP 3.0 & 4.0)**
  - Synchronization b/w parents and children — “omp taskwait”
  - Synchronization b/w siblings — “depend” clause



```
#pragma omp task depend(out: A)//T1
     {S1}
#pragma omp task depend(in: A) //T2
     {S2}
#pragma omp task // T3
     {S3}
#pragma omp taskwait // Tw
```

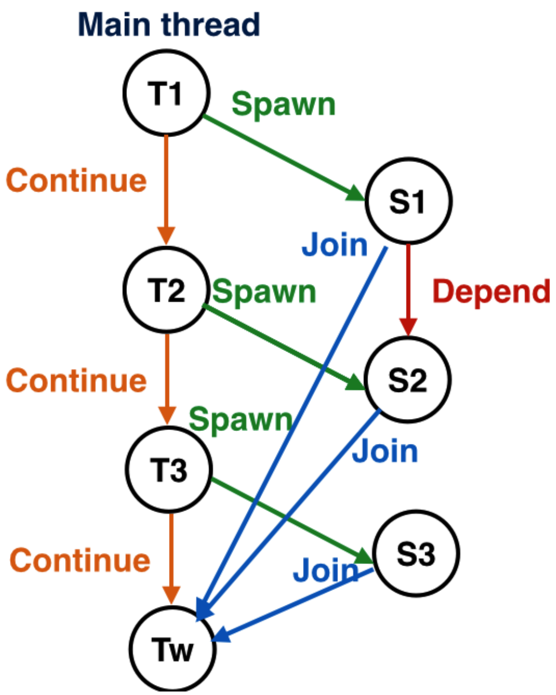


Background: Serial-Elision property

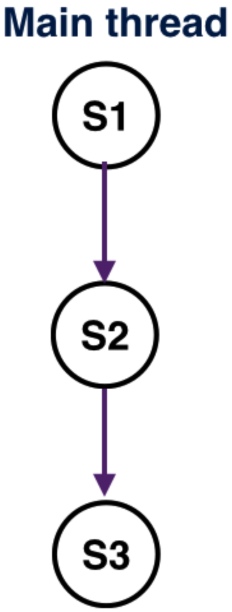
Removal of all parallel constructs results in a sequential program that is a valid (albeit inefficient) implementation of the parallel program semantics.

```
#pragma omp task depend(out: A)//T1
    {S1}
#pragma omp task depend(in: A) //T2
    {S2}
#pragma omp task // T3
    {S3}
#pragma omp taskwait // Tw
```

Original program



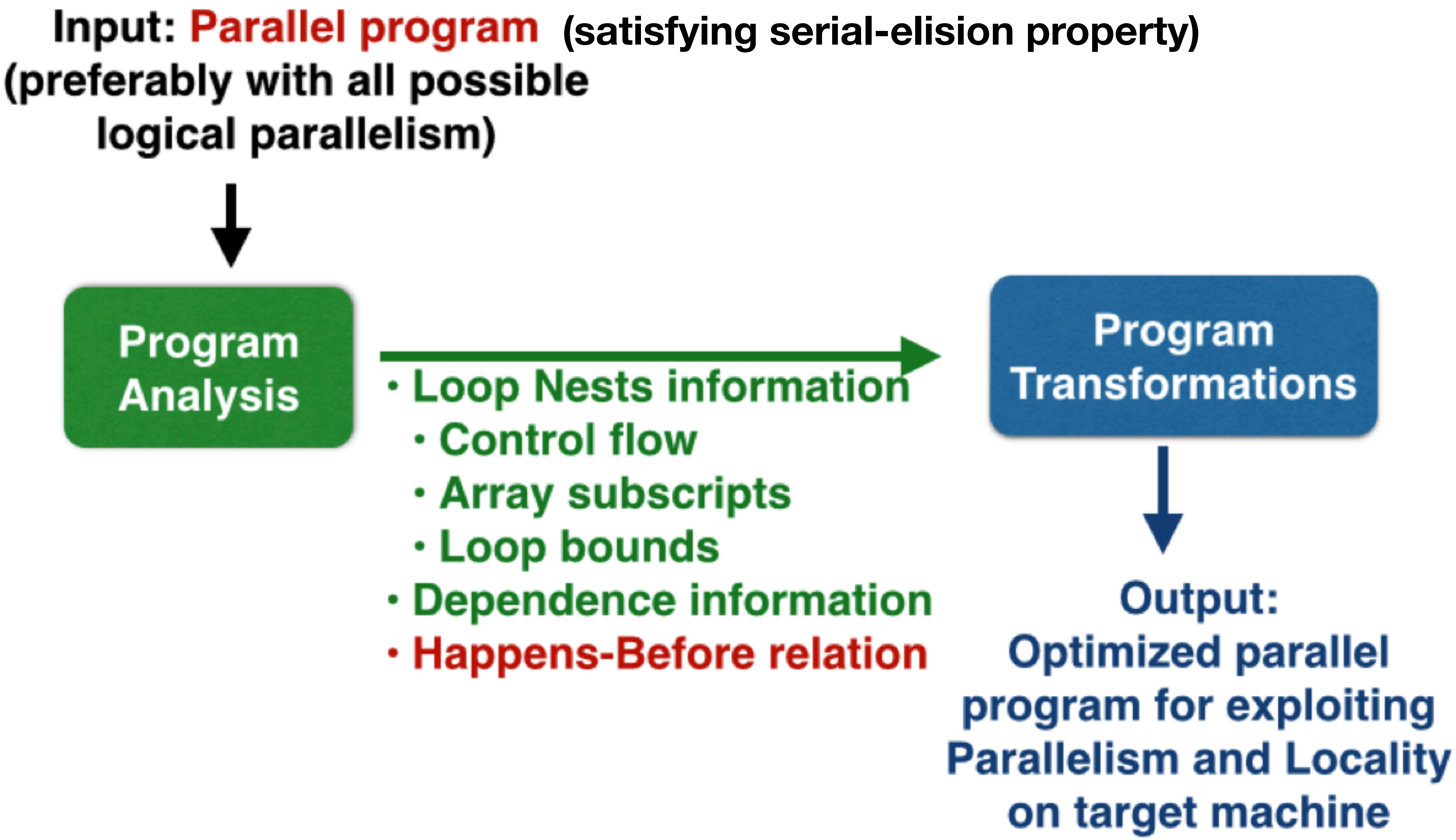
Task dependence graph of the program



Graph after removing parallel constructs

Satisfies serial-elision property

Our Approach (PoPP)



Input: Parallel program (satisfying serial-elision property)
(preferably with all possible logical parallelism)

Program Analysis

- Loop Nests information
  - Control flow
  - Array subscripts
  - Loop bounds
- Dependence information
- Happens-Before relation

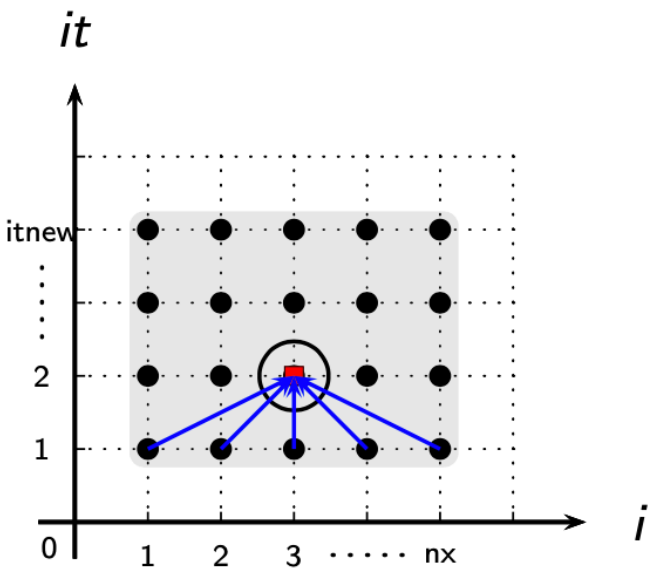
Program Transformations

Output:
Optimized parallel program for exploiting Parallelism and Locality on target machine

PoPP — Polyhedral optimizations of Parallel Programs

Step-1: Compute dependences based on the sequential order (use serial-elision and ignore parallel constructs)

```
#pragma omp parallel
#pragma omp single
{
    for (it = itold + 1; it <= itnew; it++) {
        for (i = 0; i < nx; i++) {
#pragma omp task depend(out: u[i])
    depend(in: unew[i])
            for (j = 0; j < ny; j++)
S1:             u[i][j] = unew[i][j];
        }

        for (i = 0; i < nx; i++) {
#pragma omp task depend(out: unew[i])
    depend(in: f[i], u[i-1], u[i], u[i+1])
            for (j = 0; j < ny; j++)
S2:             cpd(i, j, unew, u, f);
         }
    }
    #pragma omp taskwait
}
```

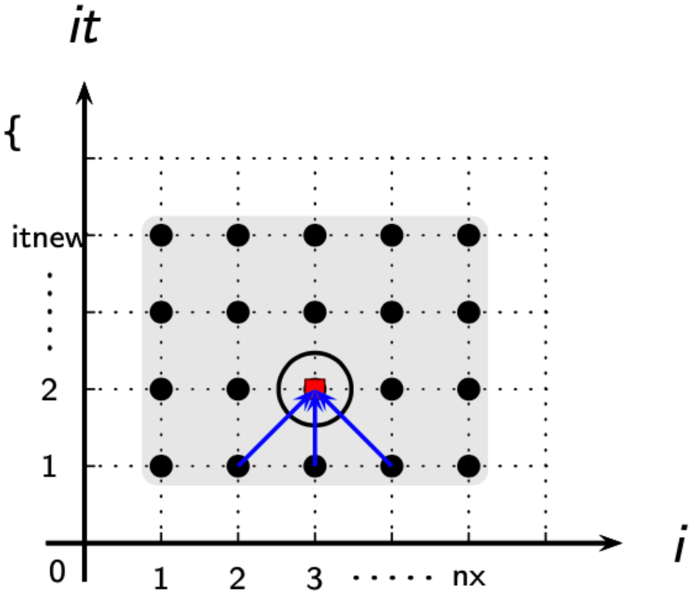


(S2 → S1)
dependences
across it & i loops

Jacobi scientific benchmark from the KASTORS suite

Step-2: Compute happens-before relations using parallel constructs (ignoring statement bodies)

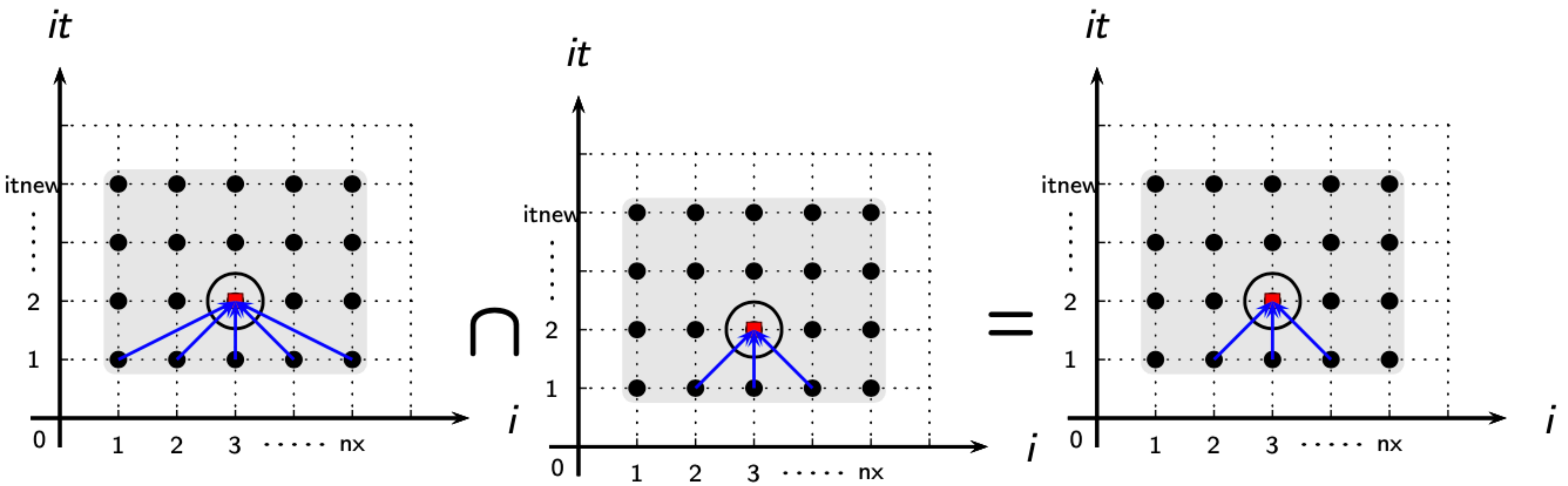
```
#pragma omp parallel
#pragma omp single
{
    for (it = itold + 1; it <= itnew; it++) {
        for (i = 0; i < nx; i++) {
#pragma omp task depend(out: u[i])
    depend(in: unew[i])
            for (j = 0; j < ny; j++)
S1:             u[i][j] = unew[i][j];
        }

        for (i = 0; i < nx; i++) {
#pragma omp task depend(out: unew[i])
    depend(in: f[i], u[i-1], u[i], u[i+1])
            for (j = 0; j < ny; j++)
S2:             cpd(i, j, unew, u, f);
         }
    }
    #pragma omp taskwait
}
```



(S2→S1) HB edges across it & i loops

Jacobi scientific benchmark from the KASTORS suite

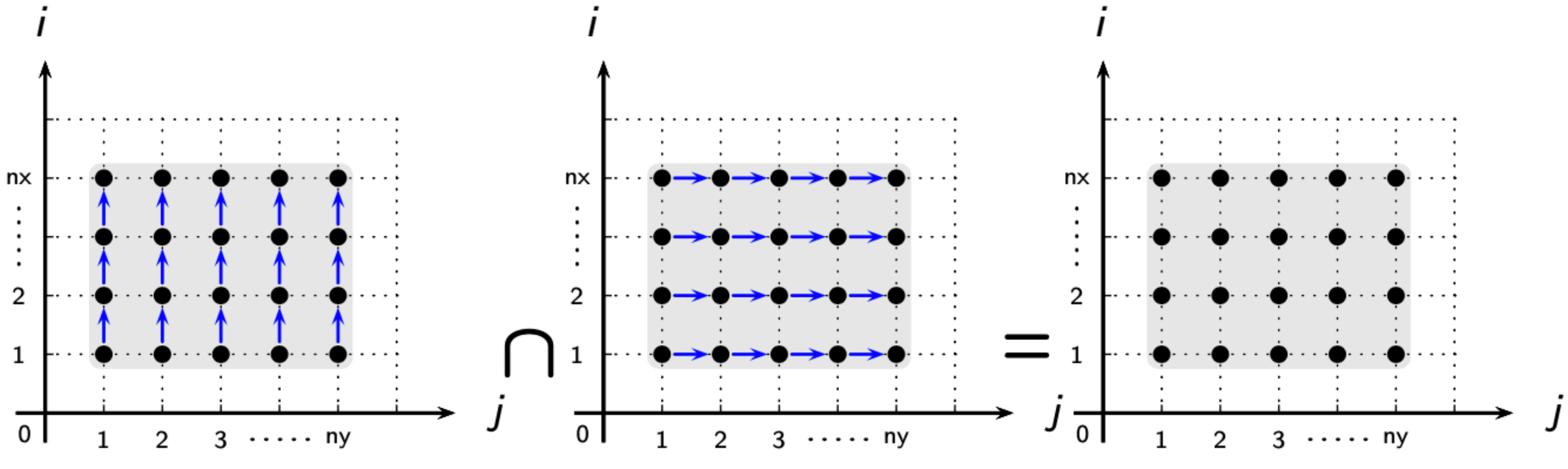
Step-3: Intersect dependences (Best of both worlds)



Conservative dependences $\mathcal{P}_1^{S2\to S1}$

HB relation $\mathcal{HB}_1^{S2\to S1}$

Refined dependences $\mathcal{P}'^{S2\to S1}_1$

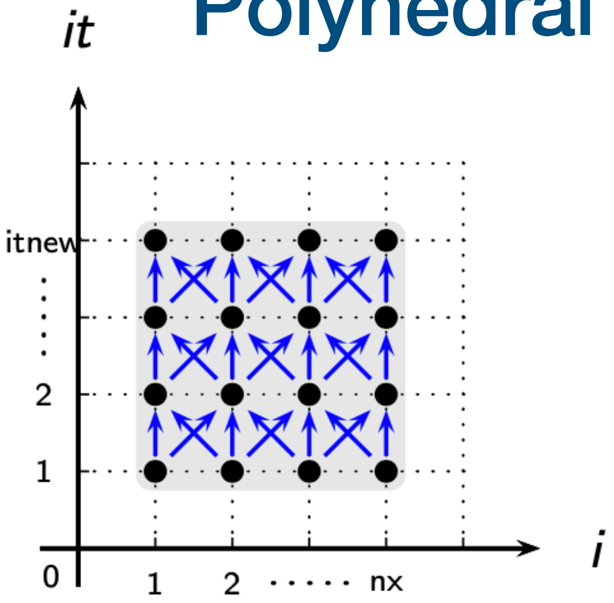


Conservative dependences $\mathcal{P}_1^{S1\to S1}$ (j-loop is parallel for S1)

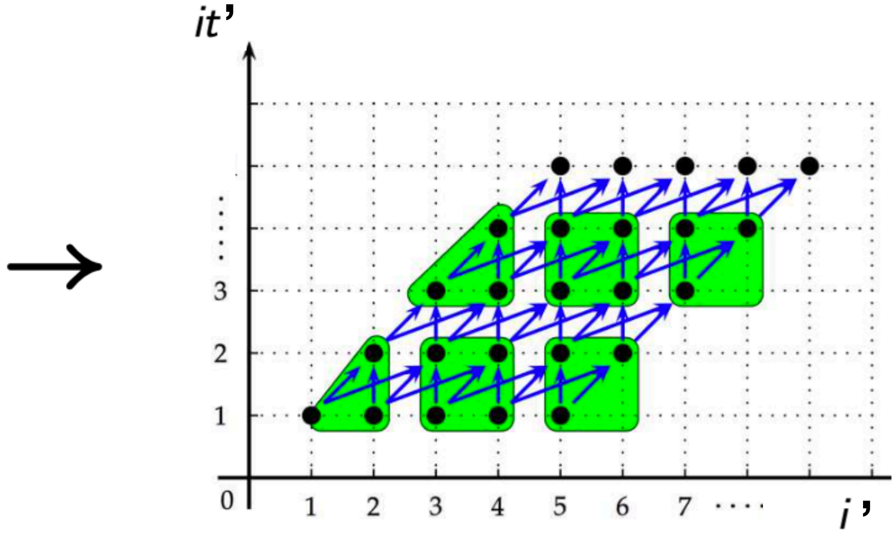
HB relation $\mathcal{HB}_1^{S1\to S1}$ (i-loop is parallel for S1)

Refined dependences $\mathcal{P}'^{S1\to S1}_1$ (No dependences for S1)

Step-4: Pass refined dependences to Polyhedral optimizers (PolyAST)

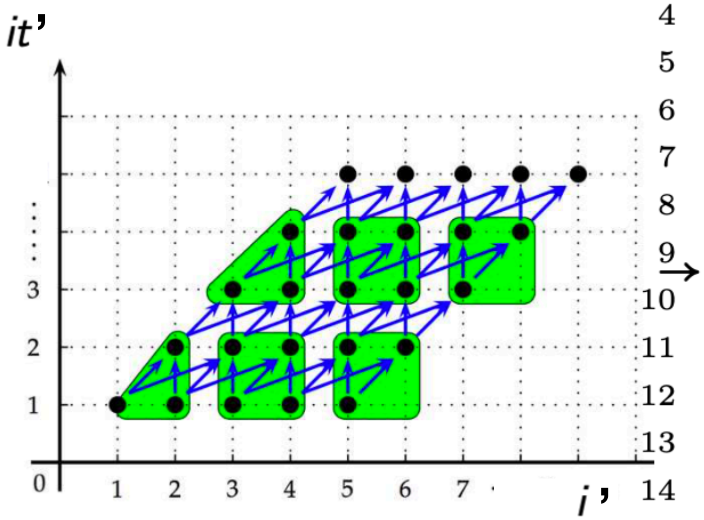


Refined dependences, $\mathcal{P}'^{S2 \rightarrow S1}_{1}$



- **Refined dependences enable a broad set of transformations**
  - i-loop is parallel, but invalid rectangular tiling
  - Skewing transformation to enable rectangular tiling

Step-5: Generate code



Omitted tiling for brevity

```
#pragma omp parallel for
    private(c3,c5) ordered(2)
for (c1 = itold + 1; c1 <= itnew; c1++) {
   for (c3 = 2 * c1; c3 <= 2 * c1 + nx; c3++) {

   #pragma omp ordered
   depend(sink: c1-1, c3) depend(sink: c1, c3-1)
       if (c3 <= 2 * c1 + nx + -1) {
           for (c5 = 0; c5 < ny; c5++)
S1:             u[-2*c1+c3][c5] = unew[-2*c1+c3][c5]
       }

       if (c3 >= 2 * c1 + 1) {
           for (c5 = 0; c5 < ny; c5++)
S2:           cpd(-2*c1+c3-1, c5, unew, u, f);
       }
   #pragma omp ordered depend(source)
 }}
```

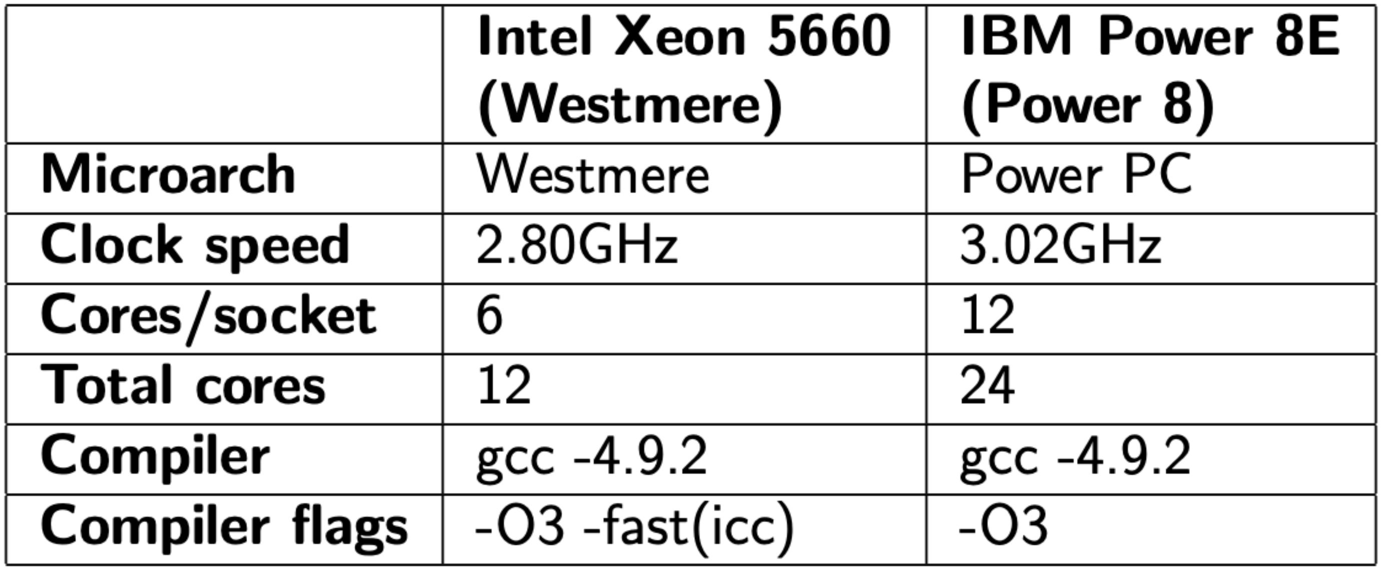
- **Invoke polyhedral code generators (PolyAST)**
  - Capable of scanning the complex iteration space
  - Fine-grained (point-to-point) synchronization instead of barriers

Evaluation

- **PoPP was implemented in ROSE source to source compiler framework and evaluated on the following benchmarks.**

- **KASTORS — Task parallel benchmarks (3)**
  - Jacobi, Jacobi-blocked, Sparse LU

- **RODINIA — Loop parallel benchmarks (8)**
  - Back propagation, CFD solver, Hotspot, Kmeans, LUD, Needleman–Wunsch, particle filter, path finder



	Intel Xeon 5660 (Westmere)	**IBM Power 8E (Power 8)**
Microarch	Westmere	Power PC
Clock speed	2.80GHz	3.02GHz
Cores/socket	6	12
Total cores	12	24
Compiler	gcc -4.9.2	gcc -4.9.2
Compiler flags	-O3 -fast(icc)	-O3

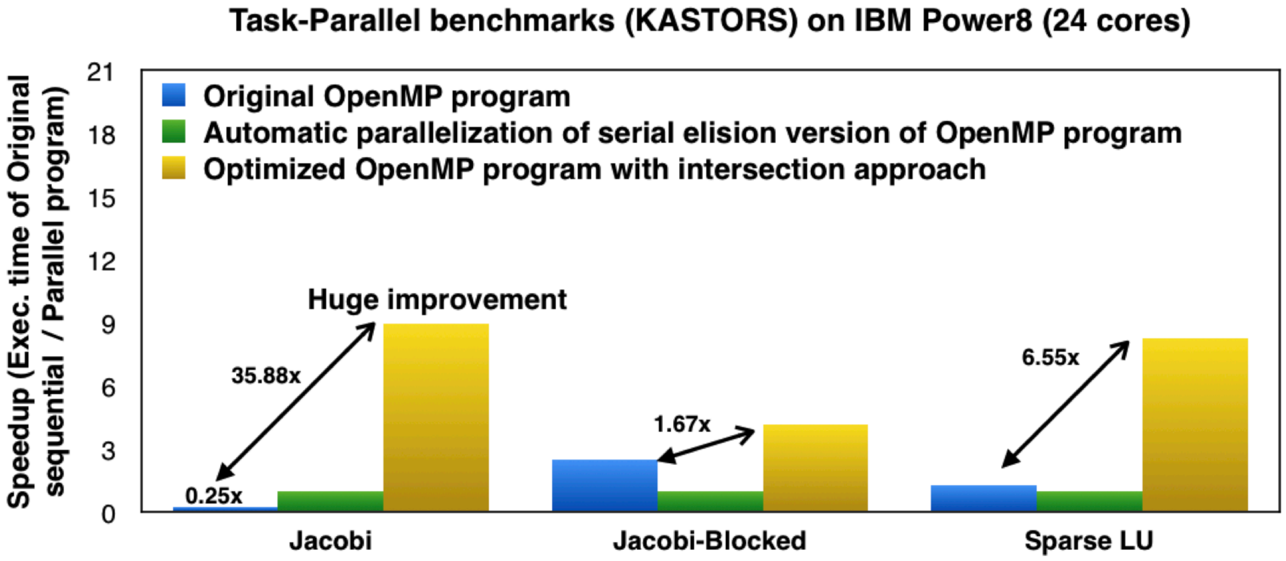
Variants

- **Original OpenMP program**
  - Written by programmer/application developer

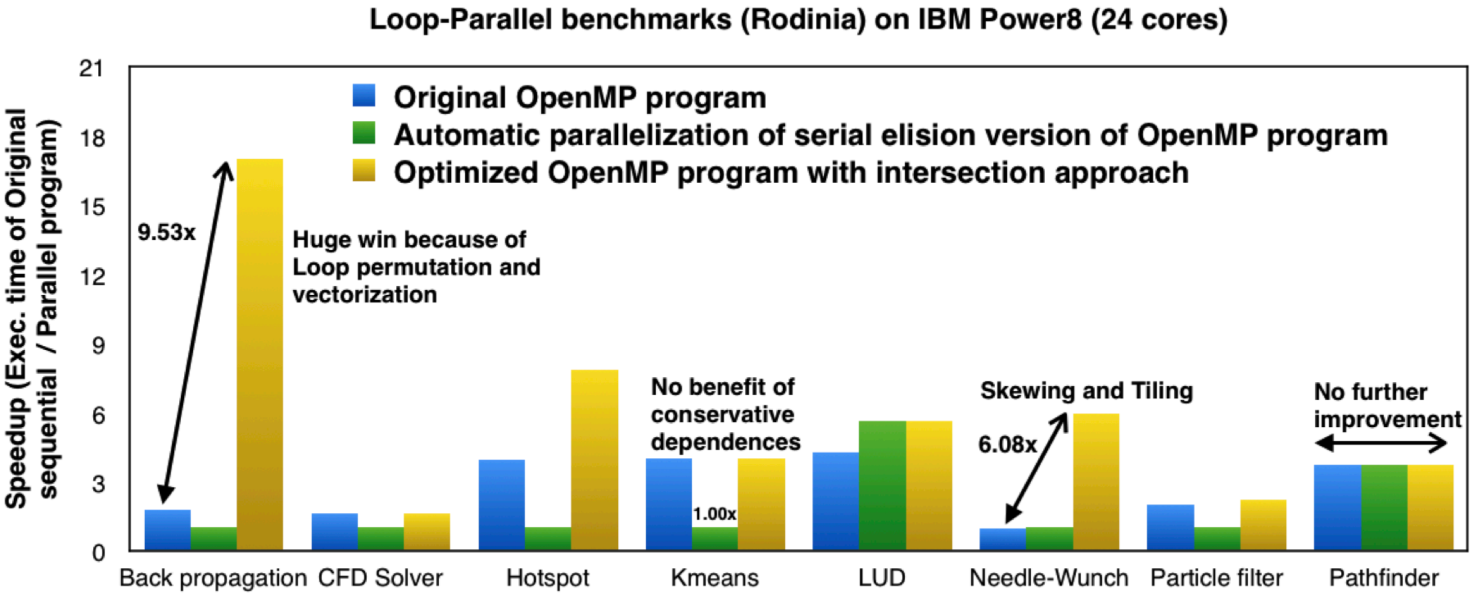
- **Automatic optimization and parallelization of serial-elision version of the OpenMP program**
  - Automatic optimizers (PolyAST)

- **Optimized OpenMP program with our approach**
  - Our framework (PoPP) which extends PolyAST with the intersection of happens- before and data dependence relations

Evaluation on IBM Power 8



Geometric mean improvement - 7.32x



Geometric mean improvement - 1.89x

Summary & Related Work

- ***Summary:***
  - Extended the foundations of optimizing compiler for analyzing parallel programs and also advanced the dependence analysis.
  - Broadened the range of applicable legal transformations
  - Geometric mean performance improvements of 1.62X on Intel westmere and 2.75X on IBM Power8
- ***Related work:***
  - Data-flow analysis of explicitly parallel programs [Yuki et al. PPoPP'13]
  - Improved loop dependence analysis for GCC auto-vectorization [Jenson et al. TACO'17]
  - Enabled classical scalar optimizations for explicitly-parallel programs using "serial-elision" property [TAPIR — Tao et al. PPoPP'17]

Key Contributions

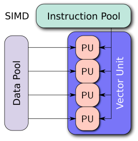
Advancing Compiler Optimizations for General-Purpose Parallel Architectures

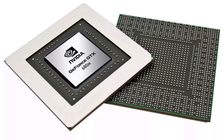
1) Analysis and optimization of explicitly parallel programs (PACT’15)

Multi-core/Many-core CPUs

2) Unification of storage transformations with loop transformations (LCPC’18)

Vector Units (SIMD, SIMT)





Advancing Compiler Optimizations for Domain-Specific Parallel Architectures

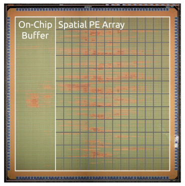
3) Domain-specific compiler for graph analytics on thread migratory hardware (MCHPC’18)

Thread migratory (EMU)

EmuTechnology

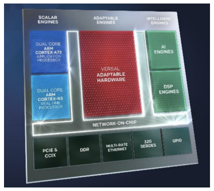
4) Data-centric compiler for DNN operators on flexible spatial accelerators (ArXiv’20)

Flexible Spatial accelerators



5) Domain-specific compiler for tensor convolutions on 2D SIMD units (Under submission)

Specialized vector units (AI Engine)



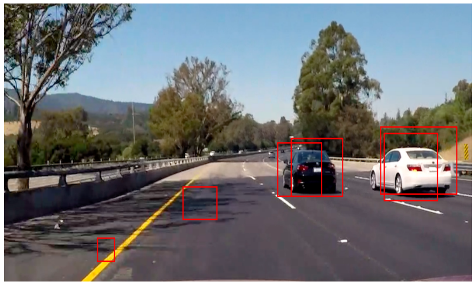
Marvel: A Data-Centric Compiler for DNN Operators onto Flexible Spatial Accelerators

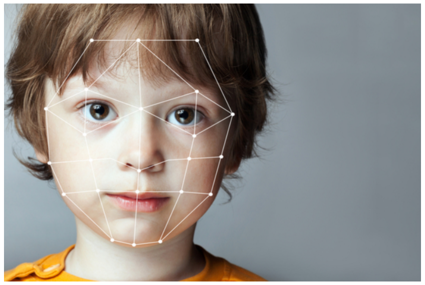
"Marvel: A Data-centric Compiler for DNN Operators on Spatial Accelerators"

Prasanth Chatarasi, Hyoukjun Kwon, Natesh Raina, Saurabh Malik, Vaisakh Haridas, Angshuman Parashar,
Michael Pellauer, Tushar Krishna, and Vivek Sarkar,

(ArXiv'20)

Deep Learning (DNN Models)

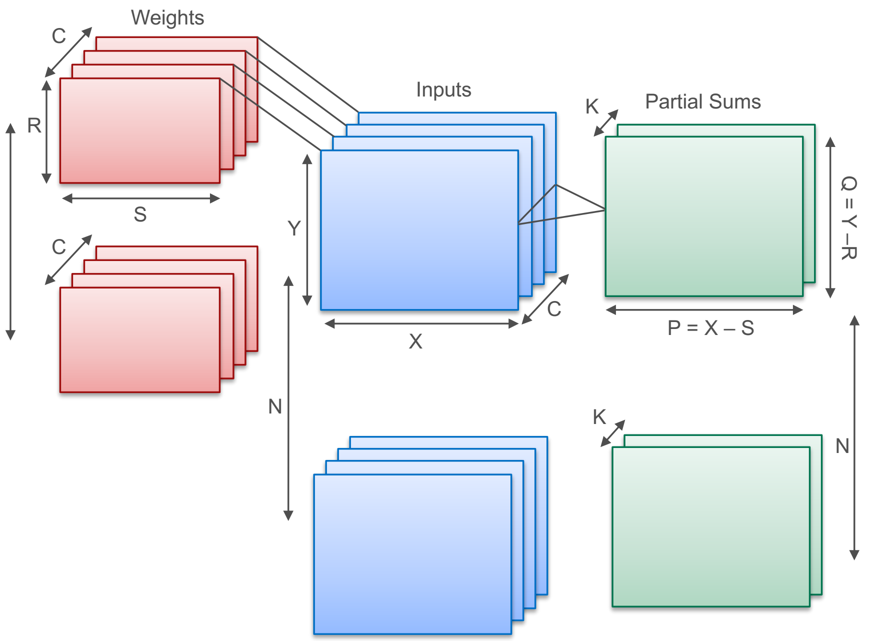




Examples of DNN Operators (Layers)

- — Regular CONV1D
- — Regular CONV2D
- — Depth-wise CONV2D
- — Transposed CONV2D
- — Regular CONV3D
- — Strided variants
- — GEMM (MatMul)
- — LSTM (RNNs)
- — Element-wise
- — Pooling
- — Fully Connected/MLP
- — …..

Regular CONV2D over 4D Tensors



Involves billions of computations

Parashar et al., ISPASS 2019

Spatial Accelerators

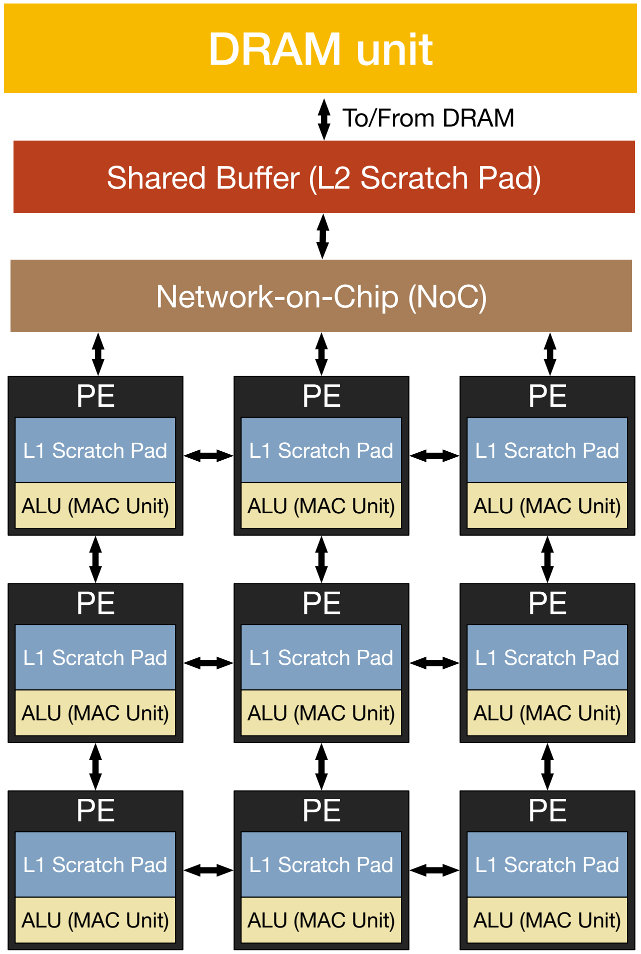
DNN Operators

— Regular CONV1D
— Regular CONV2D
— Depth-wise CONV2D
— Transposed CONV2D
— Regular CONV3D
— Strided variants
— GEMM (MatMul)
— LSTM (RNNs)
— Element-wise
— Pooling
— Fully Connected/MLP
— .....

Problem statement:

How to map for low latency, high energy efficiency?

Abstract overview



3-level accelerator

E.g., TPU, Eyeriss, NVDLA

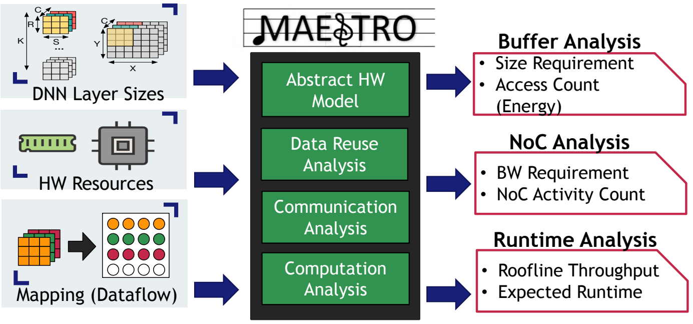
Mapping involves
1) Parallelization onto compute resources,
2) Tiling across memory resources, and
3) Exploitation of data reuse

Challenges

1. **Explosion of hardware choices in spatial accelerators**
   - Wide variety of hardware structures & data movement restrictions

2. **Rapid emergence of new DNN operators and shapes/sizes**
   - Various forms of algorithmic properties (e.g., reuses)

3. **Selection of optimized mapping from massive mapping space and also good cost models**
   - E.g., On average, O(10^{18}) mappings for CONV2D in MobileNetV2



"Understanding Reuse, Performance, and Hardware Cost of DNN Dataflows: A Data-Centric Approach"
Hyoukjun Kwon, **Prasanth Chatarasi**, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna,
In Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture (MICRO'19)

Mapping space for a 3-level accelerator

- **Multi-level tiling for memory hierarchy and for parallelization**
  - Level-1 tiling for the L1 buffer
  - Level-2 tiling for the PE array
  - Level-3 tiling for the L2 buffer

```
for(p=1; p < P; p++)
 for(s=1; s < S; s++)
  Output[p] += Weight[s] * Input[p+s]
```

(a) Plain 1D Convolution

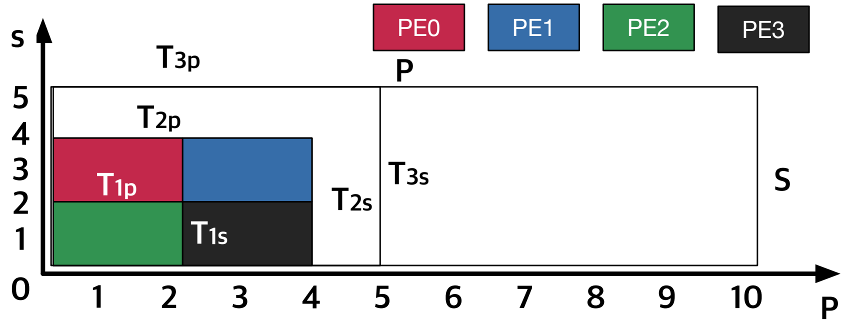
```
         for(t3p=0; t3p < P; t3p += T3p)                  ]  Level 3 Inter-tile loops
          for(t3s=0; t3s < S; t3s += T3s)                 ]
Lv3 tile - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
           for(t2p=t3p; t2p < t2p+T3p; t2p+=T2p)          ]  Level 2 Inter-tile loops
            for(t2s=t3s ; t2s < t2s+T3s; t2s+=T2s)        ]
Lv2 tile - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
             parallel_for(t1p=t2p; t1p < t1p + T2p; t1p += T1p)
              parallel_for(t1s=t1s; t1s < t1s + T2s; t1s += T1s)
Lv1 tile - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
               for(t0p=t1p; t0p < t0p + T1p; t0p +=1)
                for(t0s=t1s; t0s < t0s + T1s; t0s +=1)
                 Output[t0p] += Weight[t0s] * Input[t0p + t0s]
```

(b) Tiled 1D Convolution

- **Loop orders across tiles**
  - Inter-tile level-3 loop order
  - Inter-tile level-2 loop order

- **Data-layouts of tensors on DRAM**

- **Mapping is an unique 6D tuple in the 6-dimensional search space**

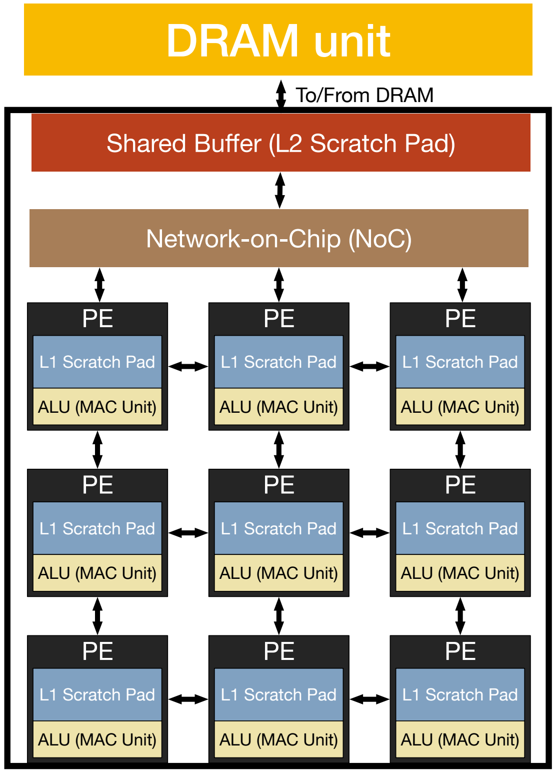


(c) Tiling Example

O(10^{18}) mappings on average for a single convolution layer in ResNet50 and MobileNetV2 models on Eyeriss-like accelerator

Our Intuition

Observation: Off-chip data movement is 2-3 orders of magnitude more expensive compared to on-chip data movement



Accessing DRAM unit: ~200x

Accessing L2 buffer: ~6x

Accessing non-local L1 buffer: ~2x

Accessing local L1 buffer: ~1x

Compute: 1x

Data movement energy

Idea: Decouple the mapping space based on off-chip and on-chip data movement, and prioritize optimizing for off-chip data movement first?

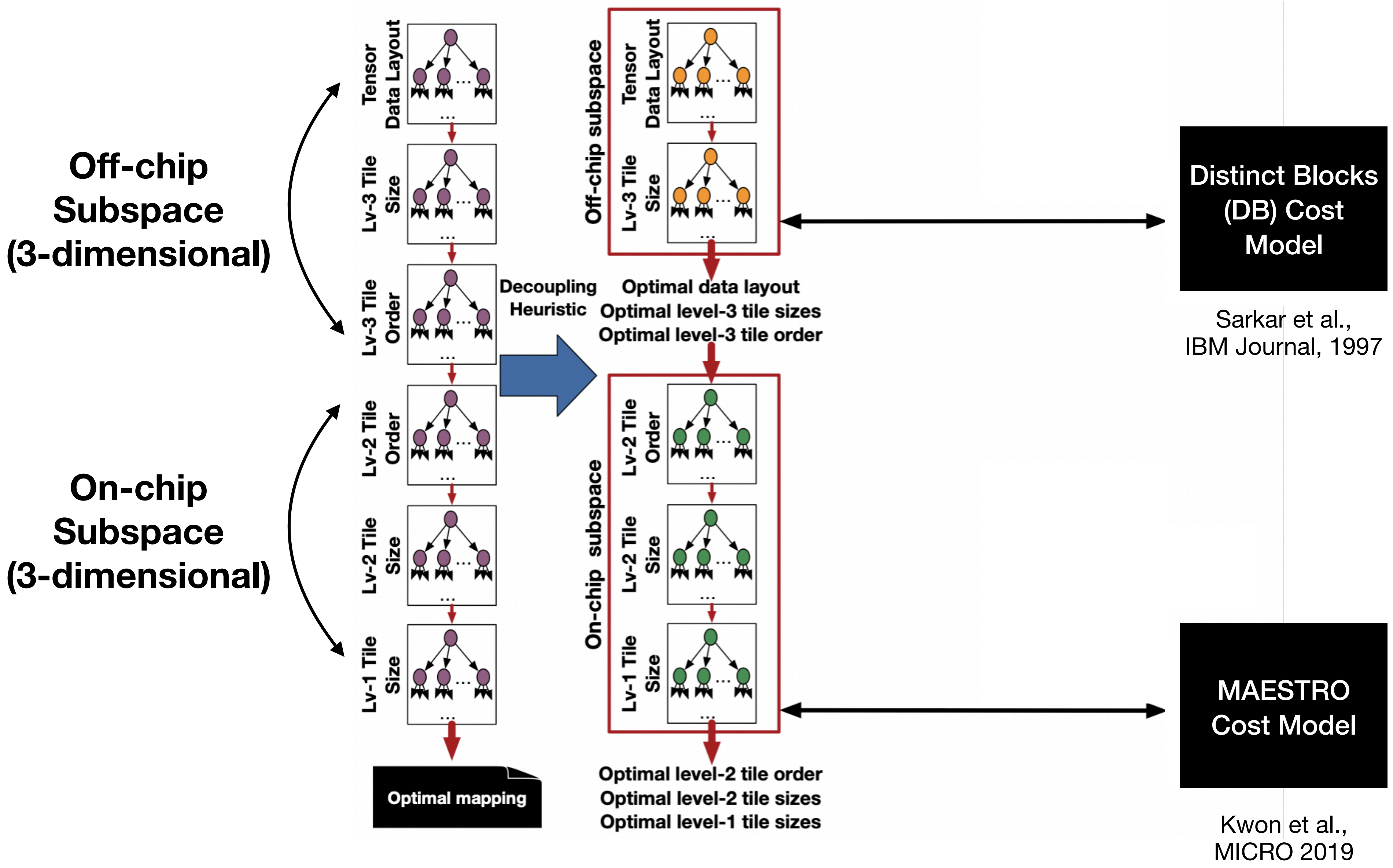
Vivienne et al., Deep Learning Tutorial

Our approach (Marvel)

Mapping space (6-dimensional)

Decoupled mapping space

Cost models



Off-chip Subspace (3-dimensional)

On-chip Subspace (3-dimensional)

Decoupling Heuristic

Optimal data layout
Optimal level-3 tile sizes
Optimal level-3 tile order

Optimal level-2 tile order
Optimal level-2 tile sizes
Optimal level-1 tile sizes

Distinct Blocks (DB) Cost Model

Sarkar et al.,
IBM Journal, 1997

MAESTRO Cost Model

Kwon et al.,
MICRO 2019

Step-1: Optimizing off-chip subspace

- **Input:** Workload and hardware configuration
- **Output:** Level-3 tile sizes & inter-tile order, and data-layouts
- **Distinct Blocks Model (DB Model)**
  - Given a parametric loop-nest and layout of tensors, the model measures distinct number of DRAM blocks for a computation tile

```
for(n=0; n<N; n++)
 for(k=0; k<K; k++)
  for(c=0; c<C; c++)
   for(p=0; p<P; p++)
    for(q=0; q<Q; q++)
     for(r=0; r<R; r++)
      for(s=0; s<S; s++)
        O[n][k][r][p] += W[k][c][r][s]
                           * I[n][c][q+r][p+s];
```

T_{3i} is the tile size for loop-i,
b is the DRAM block size

$$DB_W(T_3) \approx \left(\left\lceil \frac{T_{3S}}{b} \right\rceil\right) \times T_{3R} \times T_{3C} \times T_{3K}$$

$$DB_I(T_3) \approx \left(\left\lceil \frac{T_{3P} + T_{3S}}{b} \right\rceil\right) \times (T_{3Q} + T_{3R}) \times T_{3C} \times T_{3N}$$

$$DB_O(T_3) \approx \left(\left\lceil \frac{T_{3P}}{b} \right\rceil\right) \times T_{3Q} \times T_{3K} \times T_{3N}$$

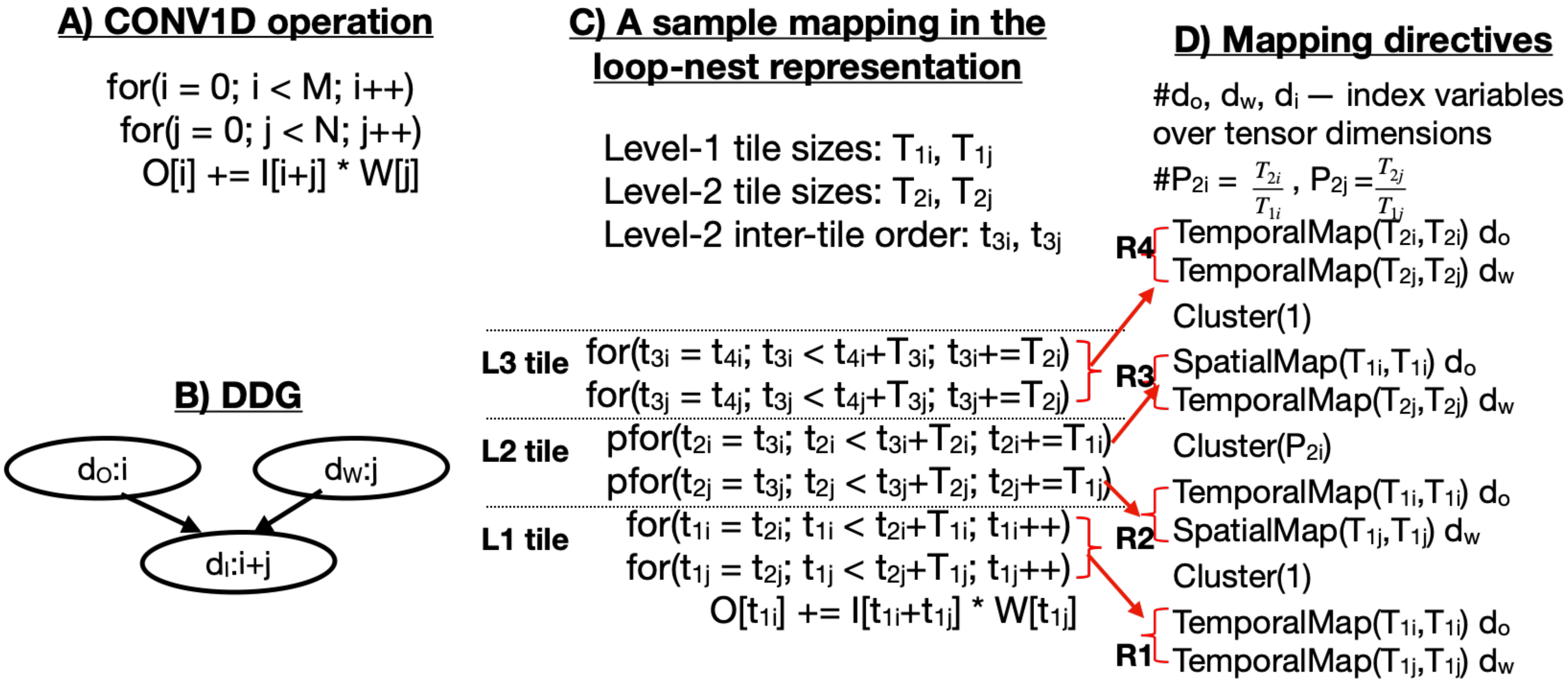
$$DMC(T_3) \approx \frac{DB_W(T_3) + DB_I(T_3) + DB_I(T_3)}{T_{3N} \times T_{3K} \times T_{3C} \times T_{3X} \times T_{3Y} \times T_{3R} \times T_{3S}}$$

$$b \times DB_{Total}(T_3) \leq \frac{|L2|}{2}$$

Sarkar et al., IBM Journal, 1997

Step-2: Optimizing on-chip subspace

- **Input:** Level-3 tile sizes, Level-3 tile order, data-layouts
- **Output:** Level-2 tile sizes, Level-2 tile order, Level-1 tile sizes
- **Iterate over each on-chip mapping, translate into MAESTRO understandable format, and invoke MAESTRO cost model**



A) CONV1D operation

```
for(i = 0; i < M; i++)
  for(j = 0; j < N; j++)
    O[i] += I[i+j] * W[j]
```

B) DDG

C) A sample mapping in the loop-nest representation

Level-1 tile sizes: T_{1i}, T_{1j}
Level-2 tile sizes: T_{2i}, T_{2j}
Level-2 inter-tile order: t_{3i}, t_{3j}

```
L3 tile  for($t_{3i}$ = $t_{4i}$; $t_{3i}$ < $t_{4i}$+$T_{3i}$; $t_{3i}$+=$T_{2i}$)
         for($t_{3j}$ = $t_{4j}$; $t_{3j}$ < $t_{4j}$+$T_{3j}$; $t_{3j}$+=$T_{2j}$)
L2 tile   pfor($t_{2i}$ = $t_{3i}$; $t_{2i}$ < $t_{3i}$+$T_{2i}$; $t_{2i}$+=$T_{1i}$)
          pfor($t_{2j}$ = $t_{3j}$; $t_{2j}$ < $t_{3j}$+$T_{2j}$; $t_{2j}$+=$T_{1j}$)
L1 tile    for($t_{1i}$ = $t_{2i}$; $t_{1i}$ < $t_{2i}$+$T_{1i}$; $t_{1i}$++)
           for($t_{1j}$ = $t_{2j}$; $t_{1j}$ < $t_{2j}$+$T_{1j}$; $t_{1j}$++)
               O[$t_{1i}$] += I[$t_{1i}$+$t_{1j}$] * W[$t_{1j}$]
```

D) Mapping directives

#d_o, d_w, d_i — index variables over tensor dimensions

#$P_{2i} = \frac{T_{2i}}{T_{1i}}$, $P_{2j} = \frac{T_{2j}}{T_{1j}}$

R4
TemporalMap(T_{2i},T_{2i}) d_o
TemporalMap(T_{2j},T_{2j}) d_w
Cluster(1)

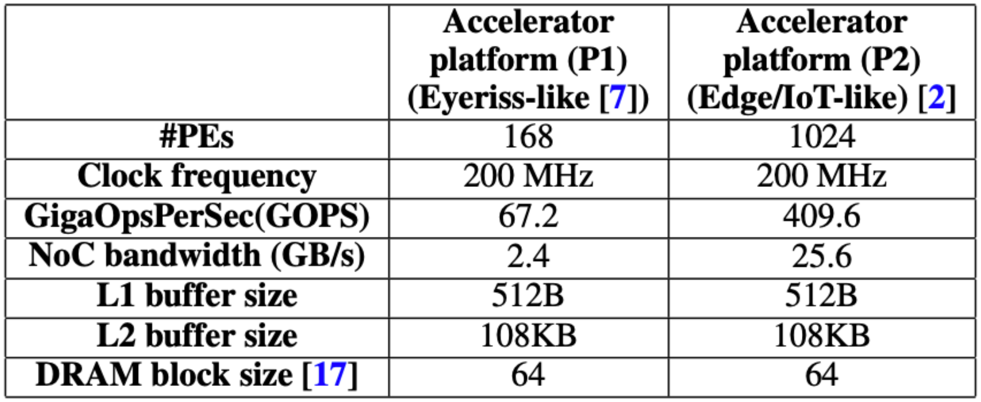
R3
SpatialMap(T_{1i},T_{1i}) d_o
TemporalMap(T_{2j},T_{2j}) d_w
Cluster(P_{2i})

R2
TemporalMap(T_{1i},T_{1i}) d_o
SpatialMap(T_{1j},T_{1j}) d_w
Cluster(1)

R1
TemporalMap(T_{1i},T_{1i}) d_o
TemporalMap(T_{1j},T_{1j}) d_w

Evaluation

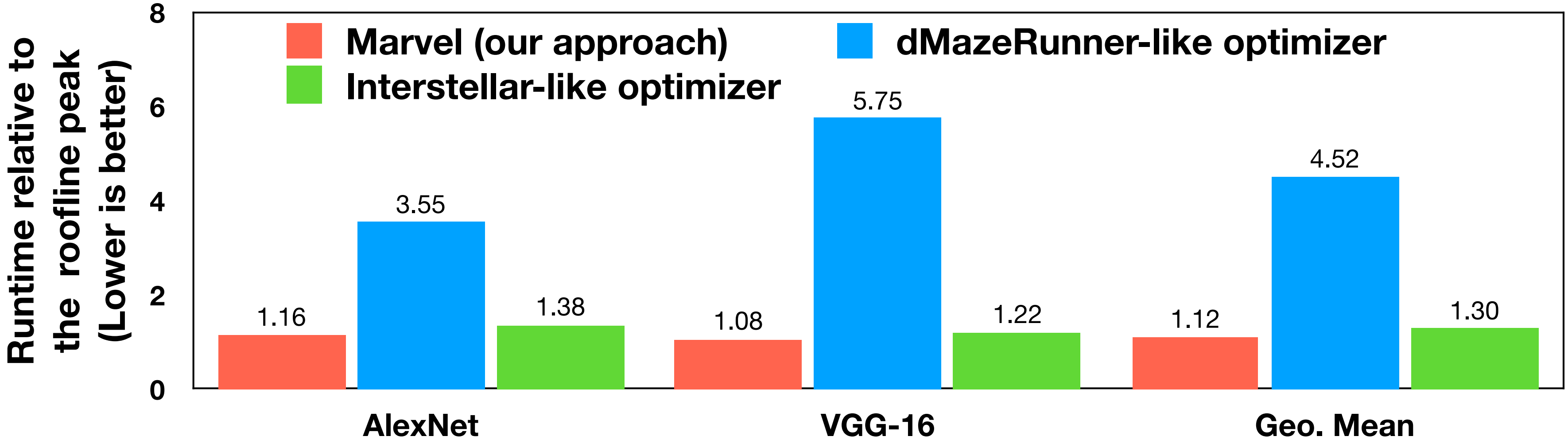
- **Four CNN models: VGG16, AlexNet, ResNet50, MobileNetV2**
  - Also, GEMM, MLP, and LSTM workloads (precision: 8bit)
- **2 representative DNN accelerators (for this talk, only P2)**



	Accelerator platform (P1) (Eyeriss-like [7])	**Accelerator platform (P2) (Edge/IoT-like) [2]**
#PEs	168	1024
Clock frequency	200 MHz	200 MHz
GigaOpsPerSec(GOPS)	67.2	409.6
NoC bandwidth (GB/s)	2.4	25.6
L1 buffer size	512B	512B
L2 buffer size	108KB	108KB
DRAM block size [17]	64	64

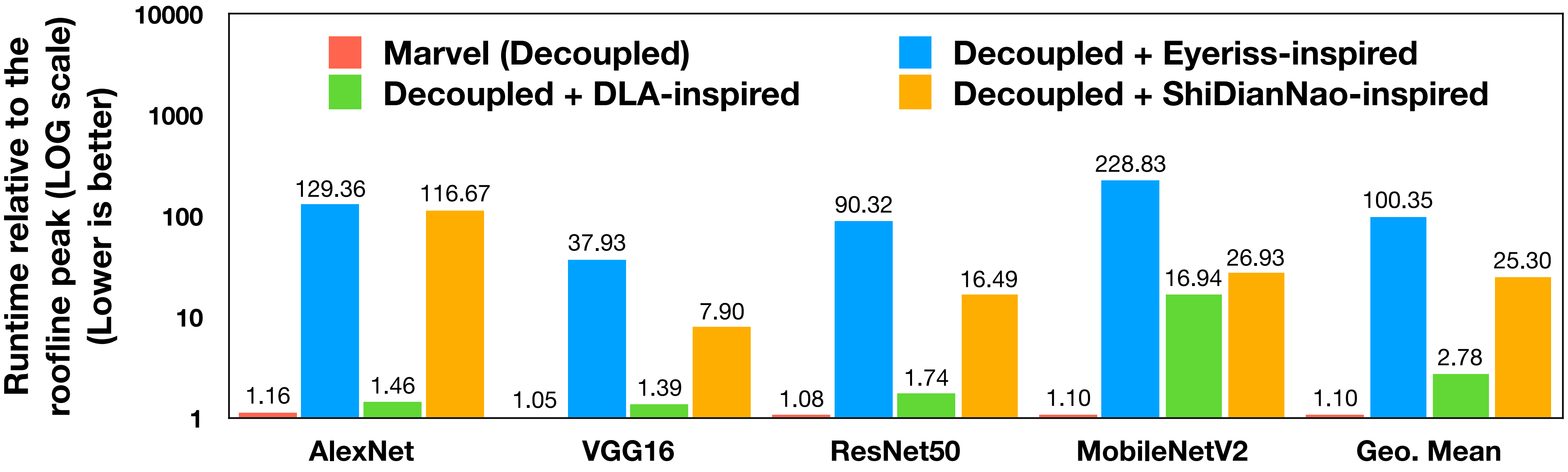
- **Comparison variants with our decoupled approach**
  - Existing optimizers: dMazeRunner, Interstellar
  - Popular on-chip mappings for CONV2D:
    - Row-stationary inspired from Eyeriss
    - Weight-stationary inspired from NVDLA,
    - Output-stationary inspired from ShiDianNao

Comparison with existing optimizers



- Evaluated other optimizers over AlexNet and VGG-16 only
  - Extremely time consuming (> 2 days) in case of MobileNetV2 and ResNet50
- *dMazeRunner-like* — Exhaustive search with aggressive pruning
  - Heavy emphasis over the batch size
- *Interstellar-like optimizer* — Parallelization across output & input channels
  - Suffers for MobileNetV2 and UNet models
- *Marvel* — Decouples the mapping space & apply pruning strategies
  - Reduce the search space on average from $O(10^{18})$ to $O(10^{8})$

Comparison with popular on-chip mappings



- *DLA-inspired mappings* — Parallelization across output & input channels
  - Good scheme except for MobileNetV2 (because of depth-wise CONV2D)
- *ShiDianNao-inspired mappings* — Parallelization across output width & height
  - Good scheme for early CONV2D layers having higher resolution
- *Marvel mappings* — Exploits > 2 levels of parallelism, various reuse orders
  - Almost close to roof-line peak (10% costlier)

Prior work on mappers

Compiler/ Mapper	Target architecture	Target goal	Accurate cost models	Operators supported/ evaluated	Level-1 Tiling	Level-2 tiling			Level-3 tiling		Approach
					Tile sizes	Parallel loops	Degree of parallelism	Inter-tile order	Tile sizes	Inter-tile order	
mRNA	MAERI	Runtime, Energy	YES	CONV2D	NA	YES	YES	YES	NO	NO	Bruteforce
TVM	VTA	Runtime	NO	CNNs	NA	YES	YES	YES	YES	YES	Annealing
DEEP MATRIX	Systolic	Runtime, Energy	YES	CONV2D, LSTM, MLP	YES	YES	YES	YES	YES	YES	Bruteforce
Zhang et al.	Flexible	Runtime	NO	CONV2D	NA	FIXED	YES	FIXED	YES	YES	Bruteforce
Ma et al.	Flexible	Runtime	NO	CONV2D	NA	FIXED	YES	FIXED	YES	YES	Bruteforce
dMaze Runner	Flexible	Runtime, Energy	YES	CONV2D	YES	FIXED	YES	FIXED	YES	FIXED	Bruteforce
Interstellar	Flexible	Runtime, Energy	YES	CONV2D, LSTM, MLP	YES	FIXED	YES	YES	YES	YES	Bruteforce
TimeLoop	Flexible	Runtime, Energy	YES	DeepBench, CNNs	YES	YES	YES	YES	YES	YES	Brute-force, random sampling
Marvel	Flexible	Runtime, Energy	YES	Any MDC Conformable	YES	YES	YES	YES	YES	YES	Decoupled

Our approach (Marvel) considers all aspects of a mapping and generate efficient latency/energy optimal mappings for flexible spatial accelerators quickly.

Summary

1. **Rapid emergence of DNN operators and hardware accelerators pose a lot of challenges to compilers**
   - Complex algorithmic reuse patterns and hardware reuse structures
   - Humongous mapping space problem

2. **Fine-grained reasoning required for mapping DNN operators to hardware accelerators for effective utilization**
   - MAESTRO cost model

3. **Effectively exploring mapping space**
   - Marvel — Proposed a decoupled off-chip/on-chip approach to efficiently explore the massive search space of mappings
   - Reduced the search space on an average by $O(10^{10})$

Key Contributions

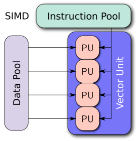
Advancing Compiler Optimizations for General-Purpose Parallel Architectures

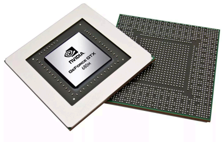
1) Analysis and optimization of explicitly parallel programs (PACT'15)

Multi-core/Many-core CPUs

2) Unification of storage transformations with loop transformations (LCPC'18)

Vector Units (SIMD, SIMT)





Advancing Compiler Optimizations for Domain-Specific Parallel Architectures

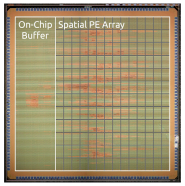
3) Domain-specific compiler for graph analytics on thread migratory hardware (MCHPC'18)

Thread migratory (EMU)

EmuTechnology

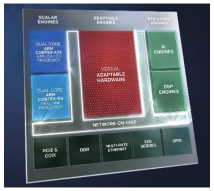
4) Data-centric compiler for DNN operators on flexible spatial accelerators (ArXiv'20)

Flexible Spatial accelerators



5) Domain-specific compiler for tensor convolutions on 2D SIMD units (Under submission)

Specialized vector units (AI Engine)



Vyasa: A High-performance Vectorizing Compiler for Tensor Convolutions onto Xilinx AI Engine

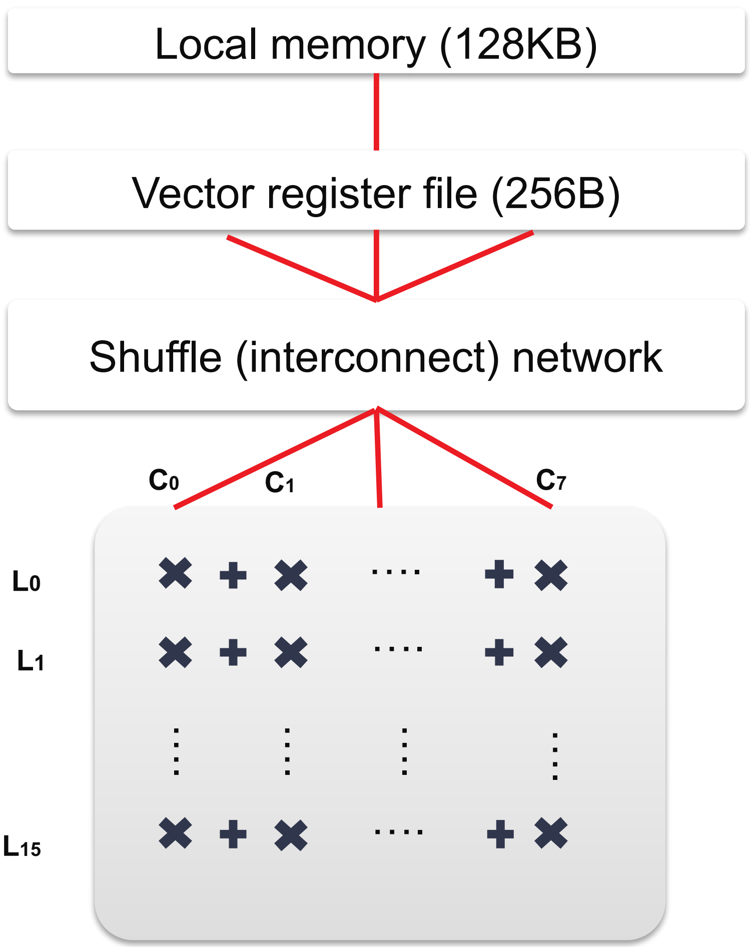
"Vyasa: A High-Performance Vectorizing Compiler for Tensor Convolutions on the Xilinx AI Engine"

Prasanth Chatarasi, Stephen Neuendorffer, Samuel Bayliss, Kees Vissers, and Vivek Sarkar

(Under submission)

Key architectural features of AI Engine

Abstract view of AI Engine



Local memory (128KB)

Vector register file (256B)

Shuffle (interconnect) network

Fixed Point SIMD Unit

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

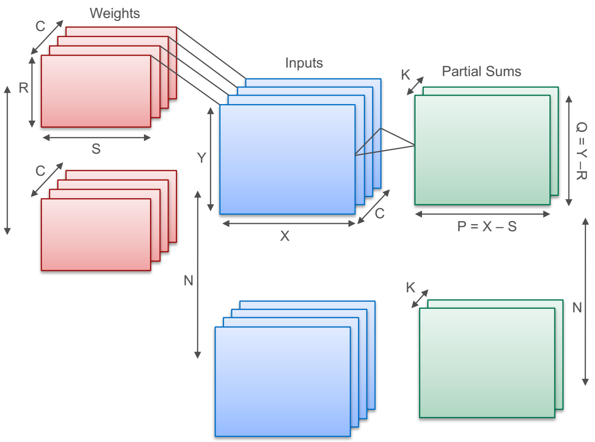
Problem Statement & Challenges

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

- Programmers manually use vector intrinsics to program 2D SIMD datapath and also explicitly specify shuffle network parameters for the data selection
- Tensor convolutions vary drastically in sizes and types
- Manually written code may not be portable to a different schedule or data-layout
- Daunting to manually explore the space of mappings

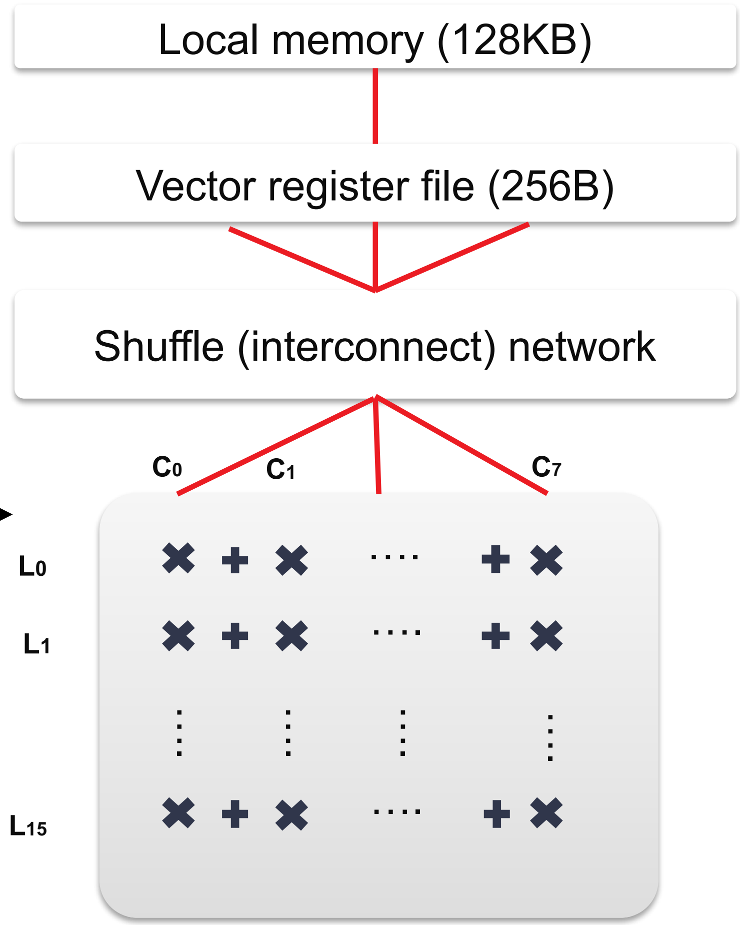
Our Compiler (Vyasa)

High-level specification of Tensor Convolutions (e.g., Halide)



- Regular CONV1D
- Regular CONV2D
- Depth-wise CONV2D
- Transposed CONV2D
- Regular CONV3D
- …..

***Vyasa*: Generating high-performance code leveraging unique capabilities**


Local memory (128KB)

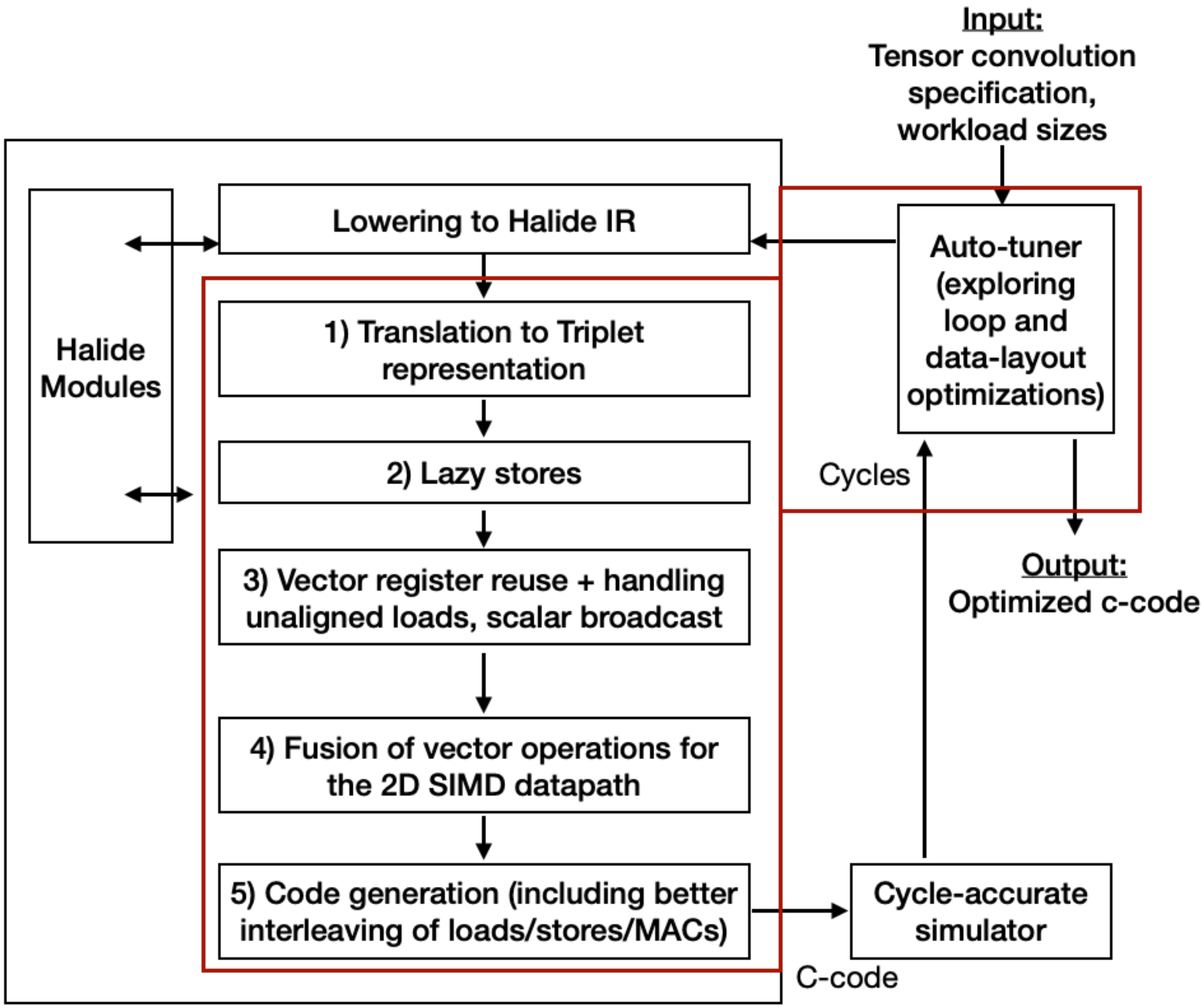
Vector register file (256B)

Shuffle (interconnect) network

Fixed Point SIMD Unit

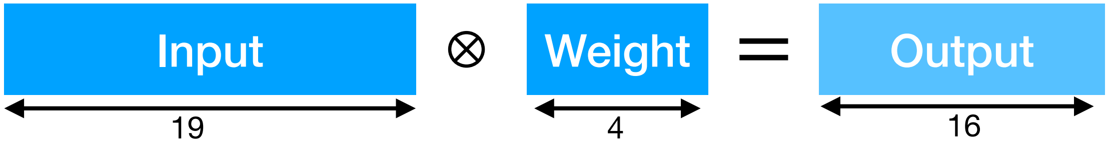
Vyasa means “compiler” in the Sanskrit language,
and also refers to the sage who first compiled the Mahabharata.

Our high-level approach (Vyasa)


Input:
Tensor convolution specification, workload sizes
Halide Modules
Lowering to Halide IR
1) Translation to Triplet representation
2) Lazy stores
3) Vector register reuse + handling unaligned loads, scalar broadcast
4) Fusion of vector operations for the 2D SIMD datapath
5) Code generation (including better interleaving of loads/stores/MACs)
Auto-tuner (exploring loop and data-layout optimizations)
Cycles
Output:
Optimized c-code
Cycle-accurate simulator
C-code

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

Running Example — CONV1D



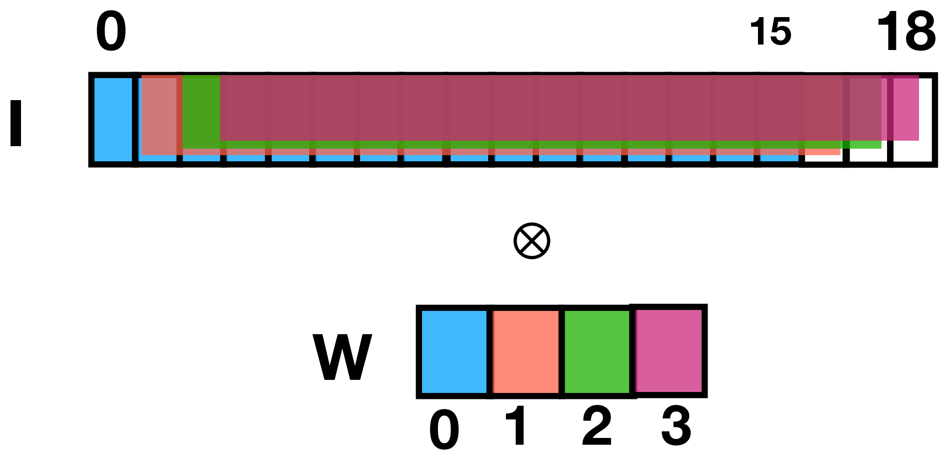
Input ⊗ Weight = Output

19 4 16

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

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

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)



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);
V3 = VMAC(V3, V4, V5);

V6 = VLOAD(I, 2:17);
V7 = BROADCAST(W, 2);
V3 = VMAC(V3, V6, V7);

V8 = VLOAD(I, 3:18);
V9 = BROADCAST(W, 3);
V3 = VMAC(V3, V8, V9);
VSTORE(O, 0:15, V3);
```

No support for unaligned loads

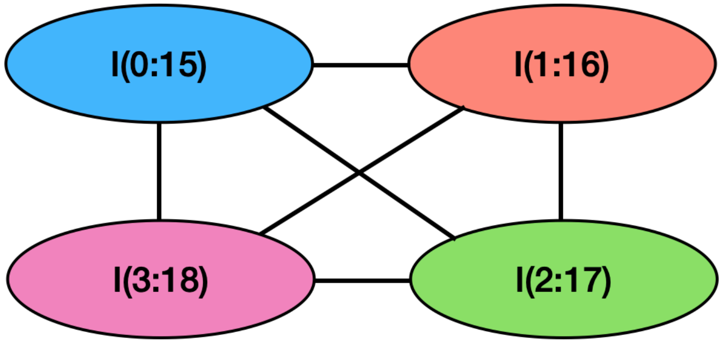
No support for broadcast operations

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

How to exploit multiple columns
of 2D vector substrate?

Exploiting Vector Register Reuse

*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)*

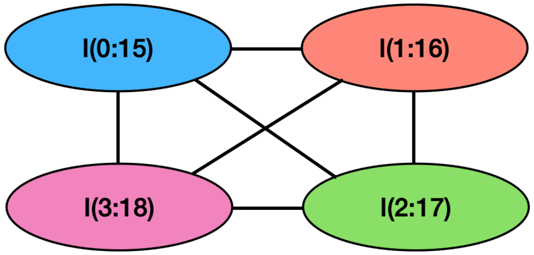


Connected component
V1 — I(0:31)

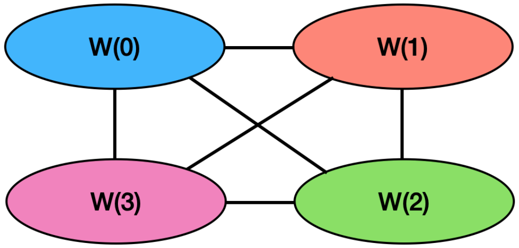
- Build "*temporal reuse graph*" with nodes being vector loads
  - Edge exists b/w nodes if there is at least one element in common
- Identify connected components
- AI Engine allows to create logical vector registers of length up to 1024 bits
  - Assign each connected component (aligned) to a logical vector register
  - Use shuffle interconnection network to select desired elements

Grouping 1D Vector Operations

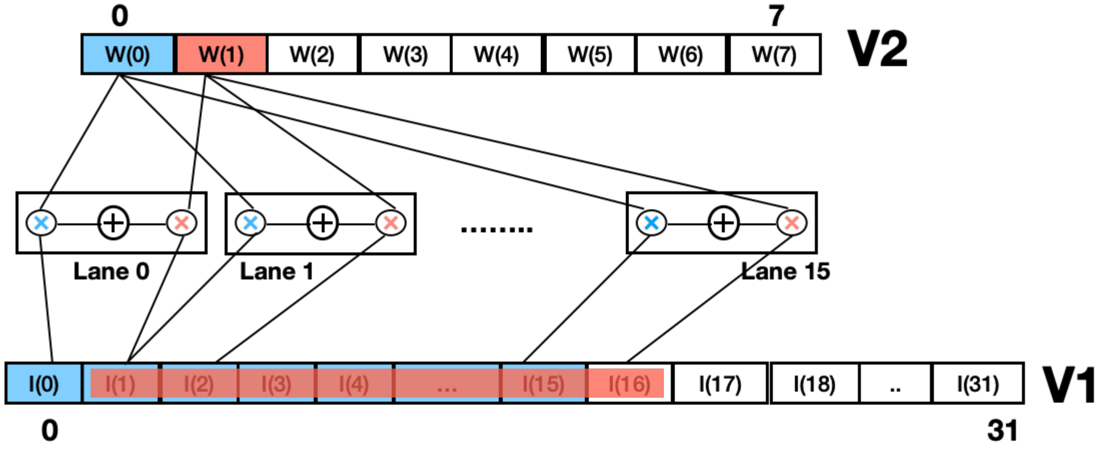
*O(0:15) += W(0) * I(0:15)*
*O(0:15) += W(1) * I(1:16)*
*O(0:15) += W(2) * I(2:17)*
*O(0:15) += W(3) * I(3:18)*



Connected component
V1 — I(0:31)



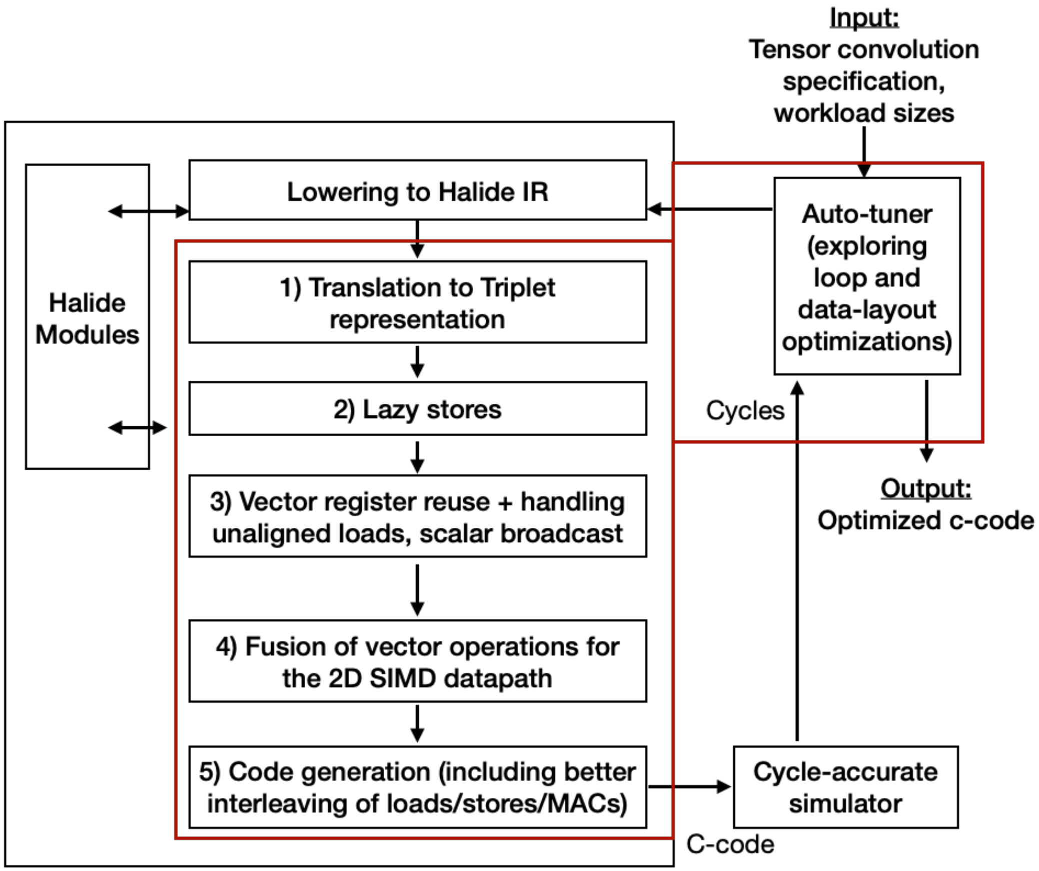
Connected component
V2 — W(0:7)



a) O(0:15) = W(0) * I(0:15) + W(1) * I(1:16)

All the 4 operations are performed with a single load of V1 and V2 (reusing maximum)

Our high-level approach (Vyasa)


Input:
Tensor convolution specification, workload sizes
Halide Modules
Lowering to Halide IR
1) Translation to Triplet representation
2) Lazy stores
3) Vector register reuse + handling unaligned loads, scalar broadcast
4) Fusion of vector operations for the 2D SIMD datapath
5) Code generation (including better interleaving of loads/stores/MACs)
Auto-tuner (exploring loop and data-layout optimizations)
Cycles
Output:
Optimized c-code
Cycle-accurate simulator
C-code

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)

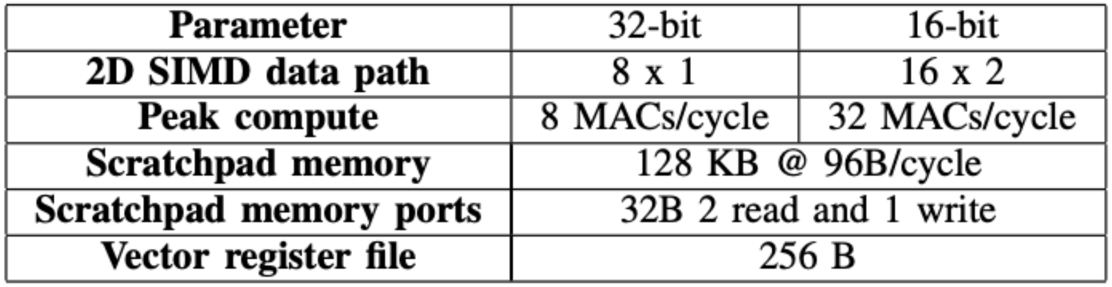
Evaluation

- **CONV2D workloads (only for this talk)**
  - CONV2D in Computer Vision (CV)

    HALIDE CODE: O(x, y) += W(r, s) * I(x+r, y+s);
  - CONV2D in DNNs

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

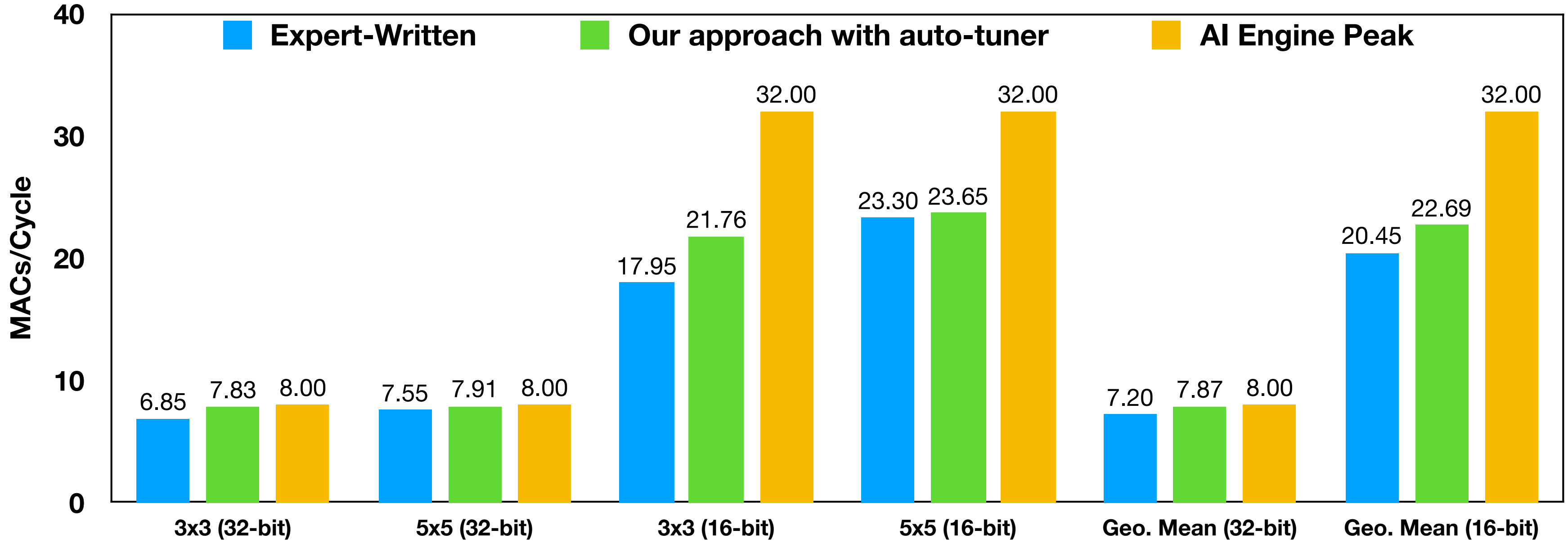
- **AI Engine setup**



Parameter	32-bit	16-bit
2D SIMD data path	8 x 1	16 x 2
Peak compute	8 MACs/cycle	32 MACs/cycle
Scratchpad memory	128 KB @ 96B/cycle	
Scratchpad memory ports	32B 2 read and 1 write	
Vector register file	256 B	

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

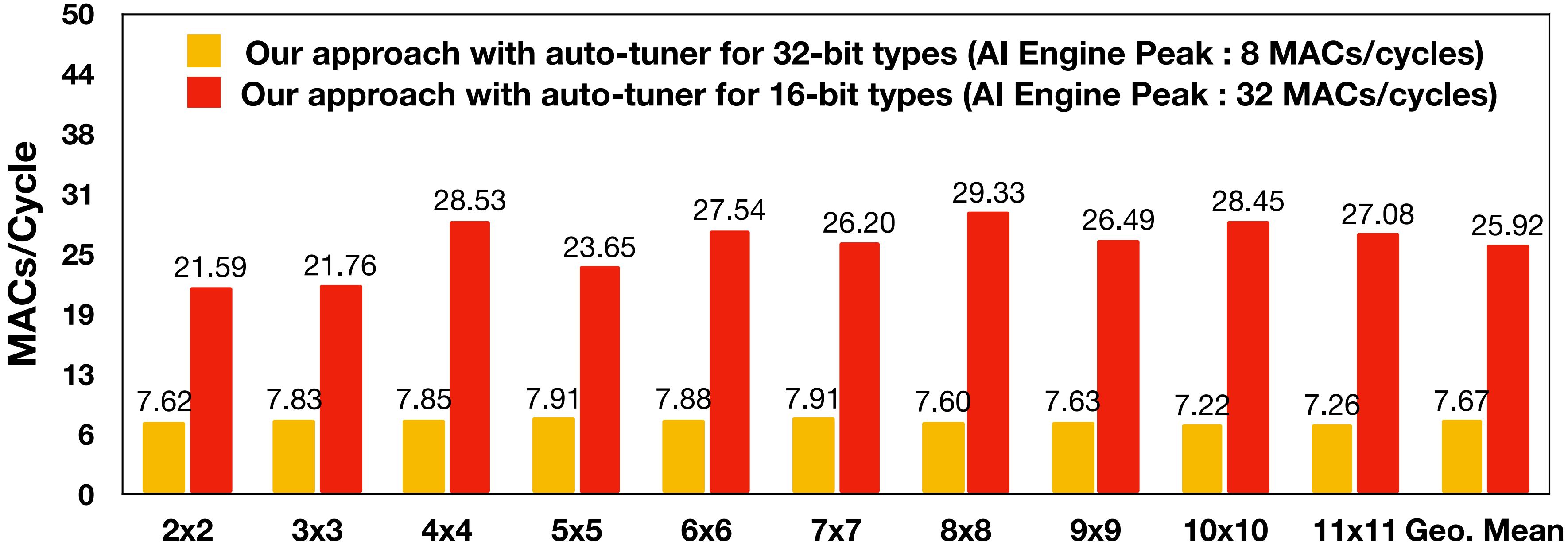
Comparison with expert-codes (CV)



- *Expert-written codes* are available only for 3x3 and 5x5 filters
  - Available as part of the Xilinx's AI Engine compiler infrastructure
  - Evaluation is over an image tile of 256x16

- *Auto-tuner was able to find better schedules*
  - Especially non-trivial unroll and jam factors

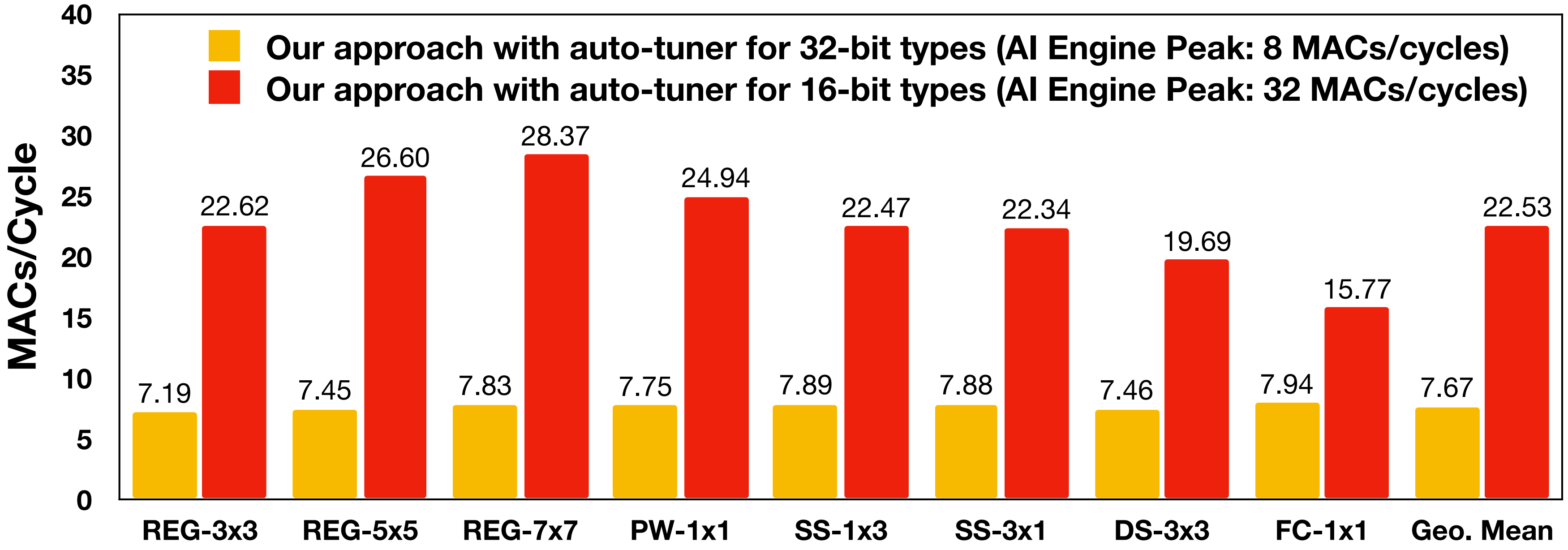
Different filter sizes in CV domain



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

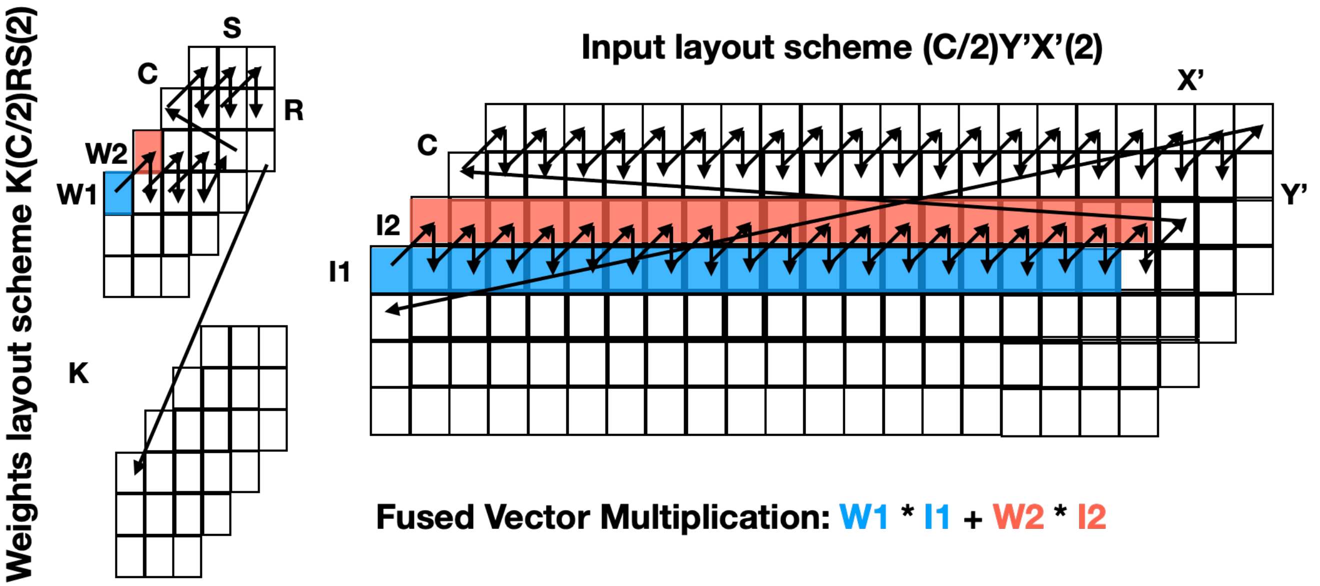
- *Odd-sized filters, our approach padded each row with an additional column*
    - For 16-bit type, number of reductions should be multiple of two (2 columns)

CONV2D's in DNN's (Batch size : 1)



- *Evaluation over an image tile of 128x2x16 (except for FC)*
- *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


Weights layout scheme K(C/2)RS(2)
S
C
R
W2
W1
K
Input layout scheme (C/2)Y'X'(2)
X'
C
Y'
I2
I1

Fused Vector Multiplication: W1 * I1 + W2 * I2

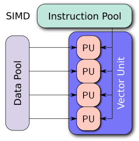
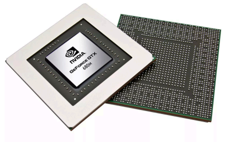
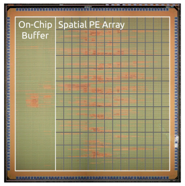
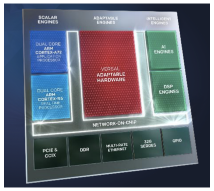
- 16-bit *REG-CONV2D (3x3)*
  - Vectorization along Output width and Reduction along Filter channels
  - For the fused vector operation (W1xI1 + W2 x I2)
    - Data for (I1, I2) should be in a single vector register for the operation
    - I1(0) and I2(0) should be adjacent for shuffle network constraints
  - (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 Related Work

- ***Summary***
  - Manually writing vector code for high-performant tensor convolutions achieving peak performance is extremely challenging!
  - **<u>Domain-specific compilation can be the key!</u>**
    - 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

- ***Related work***
  - 2D SIMD data paths and shuffle networks are unique
  - AFWK, vector unit of PEPSC architecture is the only closely related work
    - A greedy approach in their compiler to identify back to back dependent operations to map to their hardware.

Advances in compiler optimizations are critical for enabling a wide range of application domains to better exploit current and future general-purpose and domain-specific parallel architectures !!

1)	Analysis and optimization of explicitly parallel programs (PACT'15)	Multi-core/Many-core CPUs	
2)	Unification of storage transformations with loop transformations (LCPC'18)	Vector Units (SIMD, SIMT)	 SIMD Instruction Pool Data Pool PU PU PU PU Vector Unit 
3)	Domain-specific compiler for graph analytics on thread migratory hardware (MCHPC'18)	Thread migratory (EMU)	EmuTechnology
4)	Data-centric compiler for DNN operators on flexible spatial accelerators (ArXiv'20)	Flexible Spatial accelerators	
5)	Domain-specific compiler for tensor convolutions on 2D SIMD units (Under submission)	Specialized vector units (AI Engine)	 SCALAR ENGINES ADAPTABLE ENGINES INTELLIGENT ENGINES DUAL CORE ARM CORTEX-A72 APPLICATION PROCESSOR DUAL CORE ARM CORTEX-R5 REAL TIME PROCESSOR VERSAL ADAPTABLE HARDWARE AI ENGINES DSP ENGINES NETWORK-ON-CHIP PCIE & CCIX DDR MULTI-RATE ETHERNET 32G SERDES GPIO

Publications related to key contributions

1. **Prasanth Chatarasi, Stephen Neuendorffer, Samuel Bayliss, Kees A. Vissers, Vivek Sarkar; *"Vyasa: A High-Performance Vectorizing Compiler for Tensor Convolutions on the Xilinx AI Engine".* (Under submission) (2020)**

2. **Prasanth Chatarasi, Hyoukjun Kwon, Natesh Raina, Saurabh Malik, Vaisakh Haridas, Tushar Krishna, Vivek Sarkar; "*Marvel: A Data-centric Compiler for DNN Operators on Spatial Accelerators*". CoRR abs/2002.07752 (2020)**

3. **Prasanth Chatarasi, Vivek Sarkar "*A Preliminary Study of Compiler Transformations for Graph Applications on the Emu System*". MCHPC@SC 2018**

4. **Prasanth Chatarasi, Jun Shirako, Albert Cohen, Vivek Sarkar: "*A Unified Approach to Variable Renaming for Enhanced Vectorization*". LCPC 2018**

5. **Prasanth Chatarasi, Jun Shirako, Vivek Sarkar: "*Polyhedral Optimizations of Explicitly Parallel Programs*". PACT 2015**

Publications related to other contributions

6. **Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, Tushar Krishna: "*Understanding Reuse, Performance, and Hardware Cost of DNN Dataflow: A Data-Centric Approach*". MICRO 2019**

7. **Jeffrey S. Young, E. Jason Riedy, Thomas M. Conte, Vivek Sarkar, Prasanth Chatarasi, Sriseshan Srikanth: "*Experimental Insights from the Rogues Gallery*". ICRC 2019**

8. **Prasanth Chatarasi, "*Extending the Polyhedral Compilation Model for Debugging and Optimization of SPMD-style Explicitly-Parallel Programs*" [MS Thesis 2017, Rice University]**

9. **Prasanth Chatarasi, Jun Shirako, Martin Kong, Vivek Sarkar: "*An Extended Polyhedral Model for SPMD Programs and Its Use in Static Data Race Detection*". LCPC 2016**

Acknowledgments

- **Thesis committee members**
  - Dr. Vivek Sarkar (advisor), Dr. Jun Shirako (co-advisor)
  - Dr. Tushar Krishna, Dr. Santosh Pande, and Dr. Richard Vuduc

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

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

- **Habanero & Synergy Research Group Members**

- **Friends, Staff, and Family**

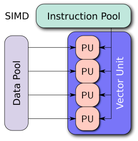
Advances in compiler optimizations are critical for enabling a wide range of application domains to better exploit current and future general-purpose and domain-specific parallel architectures !!

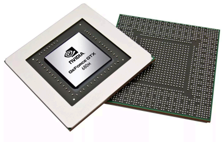
1) Analysis and optimization of explicitly parallel programs (PACT’15)

Multi-core/Many-core CPUs

2) Unification of storage transformations with loop transformations (LCPC’18)

Vector Units (SIMD, SIMT)



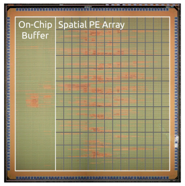


3) Domain-specific compiler for graph analytics on thread migratory hardware (MCHPC’18)

Thread migratory (EMU)

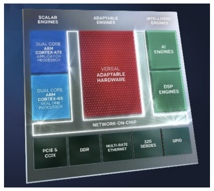
4) Data-centric compiler for DNN operators on flexible spatial accelerators (ArXiv’20)

Flexible Spatial accelerators



5) Domain-specific compiler for tensor convolutions on 2D SIMD units (Under submission)

Specialized vector units (AI Engine)


SCALAR ENGINES
ADAPTABLE ENGINES
INTELLIGENT ENGINES
DUAL CORE ARM CORTEX-A72 APPLICATION PROCESSOR
DUAL CORE ARM CORTEX-R5 REAL TIME PROCESSOR
VERSAL ADAPTABLE HARDWARE
AI ENGINES
DSP ENGINES
NETWORK-ON-CHIP
PCIE & CCIX
DDR
MULTI-RATE ETHERNET
32G SERDES
GPIO