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

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

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

```c
#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
```

![Diagram showing the execution flow of tasks T1, T2, T3, and Tw, with synchronization points S1 and S3.]
Removal of all parallel constructs results in a sequential program that is a valid (albeit inefficient) implementation of the parallel program semantics.

```c
#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
```

**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)

```c
#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)

```c
#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
    }
```
Step-3: Intersect dependences (Best of both worlds)

Conservative dependences $P_{1}^{S2 \rightarrow S1}$

HB relation $HB_{1}^{S2 \rightarrow S1}$

Refined dependences $P_{1}^{S2 \rightarrow S1}$

Conservative dependences $P_{1}^{S1 \rightarrow S1}$ (j-loop is parallel for S1)

HB relation $HB_{1}^{S1 \rightarrow S1}$ (i-loop is parallel for S1)

Refined dependences $P_{1}^{S1 \rightarrow S1}$ (No dependences for S1)
Step-4: Pass refined dependences to Polyhedral optimizers (PolyAST)

Refined dependences, $p_{i}^{S2 \rightarrow S1}$

- 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

```
#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);
        }
    }
}
```

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

<table>
<thead>
<tr>
<th></th>
<th>Intel Xeon 5660 (Westmere)</th>
<th>IBM Power 8E (Power 8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microarch</td>
<td>Westmere</td>
<td>Power PC</td>
</tr>
<tr>
<td>Clock speed</td>
<td>2.80GHz</td>
<td>3.02GHz</td>
</tr>
<tr>
<td>Cores/socket</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>Total cores</td>
<td>12</td>
<td>24</td>
</tr>
<tr>
<td>Compiler</td>
<td>gcc -4.9.2</td>
<td>gcc -4.9.2</td>
</tr>
<tr>
<td>Compiler flags</td>
<td>-O3 -fast(icc)</td>
<td>-O3</td>
</tr>
</tbody>
</table>
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

Task-Parallel benchmarks (KASTORS) on IBM Power8 (24 cores)

- Original OpenMP program
- Automatic parallelization of serial elision version of OpenMP program
- Optimized OpenMP program with intersection approach

Huge improvement

Speedup (Exec. time of Original sequential / Parallel program)

Geometric mean improvement - 7.32x

Loop-Parallel benchmarks (Rodinia) on IBM Power8 (24 cores)

- Original OpenMP program
- Automatic parallelization of serial elision version of OpenMP program
- Optimized OpenMP program with intersection approach

Huge win because of Loop permutation and vectorization

Speedup (Exec. time of Original sequential / Parallel program)

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)

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
- .....
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
- 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

\[
\begin{align*}
\text{(a) Plain 1D Convolution} & \\
\text{for}(p=1; \ p < P; \ p++) & \\
\text{for}(s=1; \ s < S; \ s++) & \\
\text{Output}[p] &= \text{Weight}[s] \times \text{Input}[p+s]
\end{align*}
\]

\[
\begin{align*}
\text{for}(t_3 = 0; \ t_3 < P; \ t_3 += T_3) & \\
\text{for}(t_2 = 0; \ t_2 < S; \ t_2 += T_2) & \\
\text{Output}[t_0] &= \text{Weight}[t_0] \times \text{Input}[t_0]
\end{align*}
\]

\[O(10^{18})\] 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

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

Accessing DRAM unit: ~200x
Accessing L2 buffer: ~6x
Accessing non-local L1 buffer: ~2x
Accessing local L1 buffer: ~1x
Compute: 1x

Data movement energy
Our approach (Marvel)

Mapping space (6-dimensional)

Decoupled mapping space

Off-chip Subspace (3-dimensional)

On-chip Subspace (3-dimensional)

Cost models

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

```plaintext
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] * l[n][c][q+r][p+s];
```

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

\[
DB_W(T_3) \approx \left(\frac{T_{3S}}{b}\right) \times T_{3R} \times T_{3C} \times T_{3K}
\]

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

\[
DB_O(T_3) \approx \left(\frac{T_{3P}}{b}\right) \times T_{3Q} \times T_{3K} \times T_{3N}
\]

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

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

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**

\[
\text{for}(i = 0; i < M; i++) \quad \text{for}(j = 0; j < N; j++) \quad O[i] += I[i+j] \times W[j]
\]

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: \(t_{3i}, t_{3j}\)

D) **Mapping directives**

\[\text{TemporalMap}(T_{2i}, T_{2j}) \quad \text{d}_o\]
\[\text{TemporalMap}(T_{2i}, T_{2j}) \quad \text{d}_w\]

Cluster(1)

\[\text{SpatialMap}(T_{1i}, T_{1j}) \quad \text{d}_o\]
\[\text{TemporalMap}(T_{2i}, T_{2j}) \quad \text{d}_w\]

Cluster(\(P_{2i}\))

\[\text{TemporalMap}(T_{1i}, T_{1j}) \quad \text{d}_o\]
\[\text{SpatialMap}(T_{1i}, T_{1j}) \quad \text{d}_w\]

Cluster(1)

\[\text{TemporalMap}(T_{1i}, T_{1j}) \quad \text{d}_o\]
\[\text{TemporalMap}(T_{1i}, T_{1j}) \quad \text{d}_w\]
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)

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

- 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

<table>
<thead>
<tr>
<th>Compiler/ Mapper</th>
<th>Target architecture</th>
<th>Target goal</th>
<th>Accurate cost models</th>
<th>Operators supported/ evaluated</th>
<th>Level-1 Tiling</th>
<th>Level-2 tiling</th>
<th>Level-3 tiling</th>
<th>Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>mRNA</td>
<td>MAERI</td>
<td>Runtime, Energy</td>
<td>YES</td>
<td>CONV2D</td>
<td>NA</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>TVM</td>
<td>VTA</td>
<td>Runtime</td>
<td>NO</td>
<td>CNNs</td>
<td>NA</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>DEEP MATRIX</td>
<td>Systolic</td>
<td>Runtime, Energy</td>
<td>YES</td>
<td>CONV2D, LSTM, MLP</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Zhang et al.</td>
<td>Flexible</td>
<td>Runtime</td>
<td>NO</td>
<td>CONV2D</td>
<td>NA</td>
<td>FIXED</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Ma et al.</td>
<td>Flexible</td>
<td>Runtime</td>
<td>NO</td>
<td>CONV2D</td>
<td>NA</td>
<td>FIXED</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>dMaze Runner</td>
<td>Flexible</td>
<td>Runtime, Energy</td>
<td>YES</td>
<td>CONV2D</td>
<td>YES</td>
<td>FIXED</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Interstellar</td>
<td>Flexible</td>
<td>Runtime, Energy</td>
<td>YES</td>
<td>CONV2D, LSTM, MLP</td>
<td>YES</td>
<td>FIXED</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>TimeLoop</td>
<td>Flexible</td>
<td>Runtime, Energy</td>
<td>YES</td>
<td>DeepBench, CNNs</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Marvel</td>
<td>Flexible</td>
<td>Runtime, Energy</td>
<td>YES</td>
<td>Any MDC Conformable</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>

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)**

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
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?

- 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)

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

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)

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

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

V1 = \text{VLOAD}(I, 0:15);
V2 = \text{BROADCAST}(W, 0);
V3 = \text{VMAC}(V1, V2);
V4 = \text{VLOAD}(I, 1:16);
V5 = \text{BROADCAST}(W, 1);
V3 = \text{VMAC}(V3, V4, V5);
V6 = \text{VLOAD}(I, 2:17);
V7 = \text{BROADCAST}(W, 2);
V3 = \text{VMAC}(V3, V6, V7);
V8 = \text{VLOAD}(I, 3:18);
V9 = \text{BROADCAST}(W, 3);
V3 = \text{VMAC}(V3, V8, V9);
\text{VSTORE}(O, 0:15, V3);

\text{No support for unaligned loads}
\text{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

- 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

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

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

All the 4 operations are performed with a single load of V1 and V2 (reusing maximum)
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 Computer Vision (CV)
    - **HALIDE CODE:** \( O(x, y) += W(r, s) \times I(x+r, y+s) \);
  - CONV2D in DNNs
    - **HALIDE CODE:** \( O(x, y, k, n) += W(r, s, c, k) \times I(x+r, y+s, c, n) \);

- **AI Engine setup**

<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>

- **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

• 16-bit \textit{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 Related Work

• **Summary**
  • Manually writing vector code for high-performant tensor convolutions achieving peak performance is extremely challenging!
  • **Domain-specific compilation 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

• **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)

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)
Publications related to key contributions


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


Publications related to other contributions


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

Acknowledgments

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

• **Collaborators**

• **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)