Next-generation compilers for emerging systems

Jeronimo Castrillon
Chair for Compiler Construction (CCC), TU Dresden,
SCADS.AI Dresden/Leipzig & Center for Advancing Electronics (cfaed) Dresden

Keynote: Workshop on Compilers, Deployment, and Tooling for Edge AI (CODAI’23)
Hamburg, Germany
September 21, 2023
Evolution of computing: Breaking walls

Transistors (thousands)

Single-core architectures

Multi-core architectures

~2005

Core count


Transistors

Performance

Frequency (MHz)

Typical power

Post CMOS, non-Von Neumann

Dark Si: specialize

Multi-core architectures

Single-core architectures
Emerging systems: Examples

High-bandwidth memory
Source: AMD, AnandTech

AI accelerators + Prog. logic
Source: AMD

Near-memory computing
Source: UPMEM

Emerging memories + in-memory computing
Source: IBM
Emerging systems: Examples

- High-bandwidth memory
- AI accelerators + Prog. logic
- Near-memory computing

Extreme heterogeneity, non Von Neumann paradigms, custom number representations, custom data mapping, complex APIs, …
Abstractions and compilation

\[ v_{ijk,e} = \sum_{i'=0}^{p} \sum_{j'=0}^{p} \sum_{k'=0}^{p} A_{kk'} A_{jj'} A_{ii'} u_{i'j'k'} \]

What we want

What we (naively) code

What compilers see

void cfd_kernel(
  double A[restrict 7][7],
  double u[restrict 216][7][7][7],
  double v[restrict 216][7][7][7])
{
  /* element loop: */
  for(int e = 0; e < 216; e++) {
    for(int i0 = 0; i0 < 7; i0++) {
      for(int j0 = 0; j0 < 7; j0++) {
        for(int k0 = 0; k0 < 7; k0++) {
          v[e][i0][j0][k0] = 0.0;
        }
      }
    }
  }
  /* end of element loop */
}
Abstractions and compilation

\[ v_{ijk,e} = \sum_{i'=0}^{p} \sum_{j'=0}^{p} \sum_{k'=0}^{p} A_{kk'} A_{jj'} A_{ii'} u_{ii'j'k'} \]

What we want

What we (naively) code

What performance experts code

```
void cfd_kernel(
    double A[restrict 7][7],
    double u[restrict 216][7][7][7],
    double v[restrict 216][7][7][7][7])
{
    // element loop: */
    for(int e = 0; e < 216; e++) {
        for(int i0 = 0; i0 < 7; i0++) {
            for(int j0 = 0; j0 < 7; j0++) {
                for(int k0 = 0; k0 < 7; k0++)
                    v[e][i0][j0][k0] = A[i0][j1] * A[j0][j1] * A[k0][k1] / u[e][i1][j1][k1];
                }        }
            }    }
    } } )  // end of element loop */
```
What we want

$\nu_{ijk,e} = \sum_{i'=0}^{p} \sum_{j'=0}^{p} \sum_{k'=0}^{p} A_{kk'} A_{jj'} A_{ii'} u_{i'j'k'} e$

Abstractions and compilation

What we want

AI accelerator

HBM+FPGA

Near-memory computing

© Prof. J. Castrillon. Keynote, CODAI 2023
Abstractions and compilation

\[ v_{ijk,e} = \sum_{i' = 0}^{p} \sum_{j' = 0}^{p} \sum_{k' = 0}^{p} A_{kk'} A_{jj'} A_{ii'} u_{i'j'k'e} \]

What we want

100X

Need for higher-level programming abstractions and next-gen compilers as well as novel computational and costs models for emerging accelerators
The power of abstractions
MLIR: Vehicle to capture abstractions

- Started by Google ~2018, now in public domain


- Not an IR, but an extensible framework
  - to describe intermediate abstractions (called dialects),
  - to optimize representations between dialects (transform, lower or raise),
  - that builds on the success of LLVM to build community/infrastructure and reuse

Source: T. Grosser, Univ. Edinburgh
**Example: Tensor expressions (Physics, ML)**

- **CFDlang**

\[ v_{ijk,e} = \sum_{i'=0}^{p} \sum_{j'=0}^{p} \sum_{k'=0}^{p} A_{kk'} A_{jj'} A_{ii'} u_{ii'j'k'e} \]

\[
\text{source} = \ldots \\
\text{var input A : matrix } \\
\text{var input u : tensorIN } \\
\text{var input output v : tensorOUT } \\
\text{var input alpha : [] } \\
\text{var input beta : [] } \\
\text{v = alpha * (A # A # A # u . } [5 8] [3 7] [1 6] )) + beta * v
\]

```c
auto A = Matrix(m, n), B = Matrix(m, n), C = Matrix(m, n);
auto u = Tensor<3>(n, n, n);
auto v = (A*B*C)(u);
```

Interpolation


Closing the performance gap

- Not really optimization magic
  - Leverage expert knowledge
  - Algebraic identities

\[
\begin{align*}
v_{ijk} &= \sum_{l,m,n} (A_{kn} \cdot (A_{jm} \cdot (A_{il} \cdot u_{lmn}))) \\
v_{ijk} &= \sum_{l,m,n} (A_{kn} \cdot A_{jm}) \cdot (A_{il} \cdot u_{lmn}) \\
v_{ijk} &= \sum_{l,m,n} (A_{kn} \cdot ((A_{jm} \cdot A_{il}) \cdot u_{lmn})))
\end{align*}
\]


Actual code variants

Easy to generate, hard to transform
Tensor intermediate language (TeIL) in MLIR

- Primitive ops instead of index maps
  - Easier to express identities (big-O trfs)
  - Uses symbolic math, infinite precision

- Specialization path to custom hardware


K. F. A. Friebel, J. Bi, J. Castrillon, "BASE2: An IR for Binary Numeral Types" in ACM HEART 2023
Domain-specific optimization

- Encode algebraic transformations
- Direct feedback to expert via DSL export

\[
t = (S \otimes (S \otimes (S \otimes u))_{cz}^{xyz})_{by}^{cxy})_{ax}^{bcx}
\]
FPGA code generation: Bus-attached FPGAs

- H2020 EU Project: Convergence HPC, Big Data and ML
- Inverse Helmholtz Kernel

\[ v_e = (S \otimes S \otimes S)D_e^{-1}(S^T \otimes S^T \otimes S^T) u_e \]
\[ t = S \# S \# S \# u . \quad [[1 \]
\[ r = D * t \]
\[ v = S \# S \# S \# r . \quad [[0 \]

Lifetime analysis
(polyhedral analysis)

Menosyne
mem-subsystem gen (buffer sharing)

Complex compilation/design flow from DSL to system-level architecture

FPGA code generation: HBM FPGA

- H2020 EU Project: Convergence HPC, Big Data and ML
- Transformations for a \textbf{17x speedup} (same precision)

FPGA code generation: HBM FPGA

- H2020 EU Project: Convergence HPC, Big Data and ML
- Variants with up to 24x better energy efficiency

<table>
<thead>
<tr>
<th></th>
<th>GFLOPS</th>
<th>Power (W)</th>
<th>Efficiency (GFLOPS/W – GOPS/W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>1.121</td>
<td>38.727</td>
<td>Alveo</td>
</tr>
<tr>
<td>Double</td>
<td>2.903</td>
<td>30.182</td>
<td>1.283</td>
</tr>
<tr>
<td>Double, $p=7.1$</td>
<td>1.257</td>
<td>26.973</td>
<td>Intel</td>
</tr>
<tr>
<td>Fixed Pt</td>
<td>1.904</td>
<td>26.299</td>
<td>1.253</td>
</tr>
<tr>
<td>Fixed Pt, $p=7$</td>
<td>2.334</td>
<td>24.731</td>
<td>1.253</td>
</tr>
<tr>
<td>Fixed Pt, $p=7$</td>
<td>0.966</td>
<td>23.849</td>
<td>1.253</td>
</tr>
<tr>
<td>Fixed Pt, $p=7$</td>
<td>0.559</td>
<td>33.987</td>
<td>1.253</td>
</tr>
<tr>
<td>Fixed Pt, $p=7$</td>
<td>0.251</td>
<td>27.949</td>
<td>1.253</td>
</tr>
</tbody>
</table>

Architectures from Domain-Specific Languages.

https://everest-h2020.eu
**Base2: Custom precision analysis**

- **Interpolation**
  \[ v_{ijk,e} = \sum_{i' = 0}^{p} \sum_{j' = 0}^{p} \sum_{k' = 0}^{p} A_{kk'} A_{jj'} A_{ii'} u_{i'j'k'e} \]

- **Significand precision** \( p \)
  (exp_bits = 6)

- **Exponent range** \( \text{Id} E \)
  (frac_bits = 32)

---

K. F. A. Friebel, J. Bi, J. Castrillon, "BASE2: An IR for Binary Numeral Types", in ACM HEART 2023
Inference: Reconfigurable HW & emerging memories
Adaptable inference

- High-level operator graphs + quantized data types (Brevitas, FINN)
- Trade-off: Pruning, early exit and confidence threshold

Adaptable inference: Results

- Trade-off space @ design time for CNVW2A2 on CIFAR10

- Combined effect of jointly adapting pruning and early exits

Offloading to the edge

- Balance local and edge compute: Accuracy, throughput, energy
  - Edge: Higher confidence, more parallelism, communication overhead
  - Device: Local compute, resource constrained
- Metrics vary with scenario: Video decoding stable (walking) or instable (tram)
Higher abstractions and data layout

- The case of racetrack memories (RTMs)
  - Density of DRAM & size/latency of SRAM!
  - Memory cell stores up to 100 bit sequentially (in tracks)
  - Latency highly depends on allocation and address traces

Tensor contractions on RTMs

- Consecutive accesses can be pre-shifted
- Zig-zagging: Avoid “rewinding the tape”
Latency comparison vs SRAM

- Un-optimized and naïve mapping: Even worse latency than SRAM
- **24% faster** (even with very conservative circuit simulation)

A. A. Khan, et al., “Optimizing Tensor Contractions for Embedded Devices with Racetrack Memory Scratch-Pads”, LCTES’19, pp. 5-18, 2019
Higher savings due to less leakage power

74% less energy (in addition to savings due to DRAM placement)
Generalization: Optimizations for RTM

- Average improvements in performance (~20%) and energy consumption (~40%)

Random forests: Irregular access patterns

- RFs Popular for decision making @ the edge
- Use training statistics for tree placement

C. Hakert, “BLOwing Trees to the Ground: Layout Optimization of Decision Trees on Racetrack Memory”, DAC 2021
Near and in-memory computing
Rich landscape of designs

- Near-memory: Processors, logic close to memory
- In-memory (aka processing using memory): Leverage device properties

**Samsung**, Lee, Sukhan, et al. ISCA 2021


**CAM accelerators**: Hu, Sharon, et al. 2021 IEDM
CINM: Generalized MLIR infrastructure

- From linear algebra abstractions (common to ML frameworks and beyond)
- Intermediate languages for **in and near memory computing**
- **Pattern recognition**, target-specific models and optimizations

A. Khan et al, "CINM (Cinnamon): A Compilation Infrastructure for Heterogeneous Compute In-Memory and Compute Near-Memory Paradigms", arXiv, Aug 2023
CINM: Generalized MLIR infrastructure

- From linear algebra abstractions (common to ML frameworks and beyond)
- Intermediate languages for in and near memory computing
- Pattern recognition, target-specific models and optimizations

---

A. Khan et al, "CINM (Cinnamon): A Compilation Infrastructure for Heterogeneous Compute In-Memory and Compute Near-Memory Paradigms", arXiv, Aug 2023
def mm(int32(64, 64) A, int32(64, 64) B) -> (int32(64, 64) C) {
    C(i,j) += A(i,k) * B(k,j)
    where i in 0:64, k in 0:64, j in 0:64
}

uint32_t mram_base_addr_A = (uint32_t) (DPU_MRAM_HEAP_POINTER);
uint32_t mram_base_addr_B = (uint32_t) (DPU_MRAM_HEAP_POINTER + ROWS * COLS *
sizeof(T));
uint32_t mram_base_addr_C = (uint32_t) (DPU_MRAM_HEAP_POINTER + 2 * ROWS * COLS
* sizeof(T));
for(int i = (tasklet_id * point_per_tasklet); i < (tasklet_id+1)*point_per_tasklet; i++) {
    if( new_row != row ){
        mram_read((__mram_ptr void const*) (mram_base_addr_A + mram_offset_A),
cache_A, COLS * sizeof(T));
    }
    mram_read((__mram_ptr void const*) (mram_base_addr_B + mram_offset_B),
cache_B, COLS * sizeof(T));
dot_product(cache_C, cache_A, cache_B, number_of_dot_products);
    ...}
    mram_write( cache_C, (__mram_ptr void *) (mram_base_addr_C + mram_offset_C),
point_per_tasklet * sizeof(T));
}
UPMEM example: Results

Matmult

Execution time (sec, log scale)

1-DIMM  128 DPUs
5-DIMMs  640 DPUs
10-DIMMs  1280 DPUs

6.1×, (1 DIMM)
21.3× (5 DIMM) and
30.4× (10 DIMM) wrt host CPU
def contr(int16(K, L, M) A, int16(L, K, N) B)  
    -> (int16(M, N) C)
{
    C(m, n) += A(k, l, m) * B(l, k, n)
}

\[\downarrow\] lowers to

%0 = linalg.transpose(%A, \{2, 0, 1\})
%1 = linalg.transpose(%B, \{1, 0, 2\})
%2 = linalg.reshape(%0, \{0, \{1, 2\}\})
%3 = linalg.reshape(%1, \{\{0, 1\}, 2\})

// eligible for offloading to CIM
linalg.matmul(%2, %3, %C)

// loop interchanged GEMM
scf.for %k = %c0 to %numTiles step %c1 {
    scf.for %j = %c0 to %tiledCols step %c1 {
        %tileB = cim.copyTile(%B, %k, %j)
        cim.write(%id, %tileB)
        scf.for %i = %c0 to %tiledRows step %c1 {
            %tileC = cim.copyTile(%C, %i, %j)
            ...  
            cim.storeTile(%tileC, %C, %i, %j)
        }
    }
}

linalg.matmul(%A, %B, %C)  

\[\downarrow\] lowers to

// tiled GEMM in the CIM dialect
%0 = constant 0 : i32
%1 = constant 1 : i32
%id = constant 0 : i32 // tile id
scf.for %i = %c0 to %tiledRows step %c1 {
    scf.for %j = %c0 to %tiledCols step %c1 {
        %tileC = cim.copyTile(%C, %i, %j)
        %tempTile = cim.allocDuplicate(%tileC)
        scf.for %k = %c0 to %numTiles step %c1 {
            %tileA = cim.copyTile(%A, %i, %k)
            %tileB = cim.copyTile(%B, %k, %j)
            cim.write(%id, %tileB)
            cim.matmul(%id, %tileA, %tempTile)
            cim.barrier(%id)
            // tileC += tempTile
            cim.accumulate(%tileC, %tempTile)
        }
    }
}

cim.storeTile(%tileC, %C, %i, %j)
Optimization results: Crossbars beyond matmult
Content addressable memories (CAMs)

- NVM-based CAMs: Great for KNNs, One-shot learning, ...
- CINM support for similarity and CAM arch exploration
- Automatic flow from TorchScript matches manual designs

Summary

- Next generation programming for extreme heterogeneity
  - Domain-specific abstractions, compilation flows, ...
  - Reconfigurable HW, HBM, data placement, near and in-memory computing

- Challenges
  - Understanding and modeling primitives from down below
  - Simulators, prototypes in interdisciplinary research efforts
  - Optimization/DSE: ML? simpler heuristics useful again?
  - Joint work across stack layers will be key!
Thanks! & Acknowledgements

..., and previous members of the group (Norman Rink, Sven Karol, Sebastian Ertel, Andres Goens), and collaborators (J. Fröhlich, I. Sbalzarini, A. Cohen, T. Grosser, T. Hoefler, H. Härtig, H. Corporaal, C. Pilato, S. Parkin, P. Jääskeläinen, J-J. Chen, A. Jones, X.S. Hu)
References


[CINM’23] A. Khan et al, "CINM (Cinnamon): A Compilation Infrastructure for Heterogeneous Compute In-Memory and Compute Near-Memory Paradigms", arXiv, Aug 2023