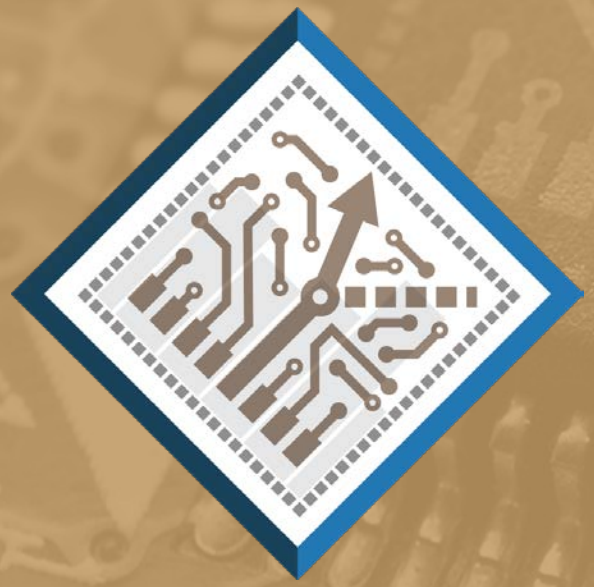
SAMAN AMARASINGHE

PROFESSOR & ASSOC. DEPT. HEAD
DEPT. OF EECS, MIT

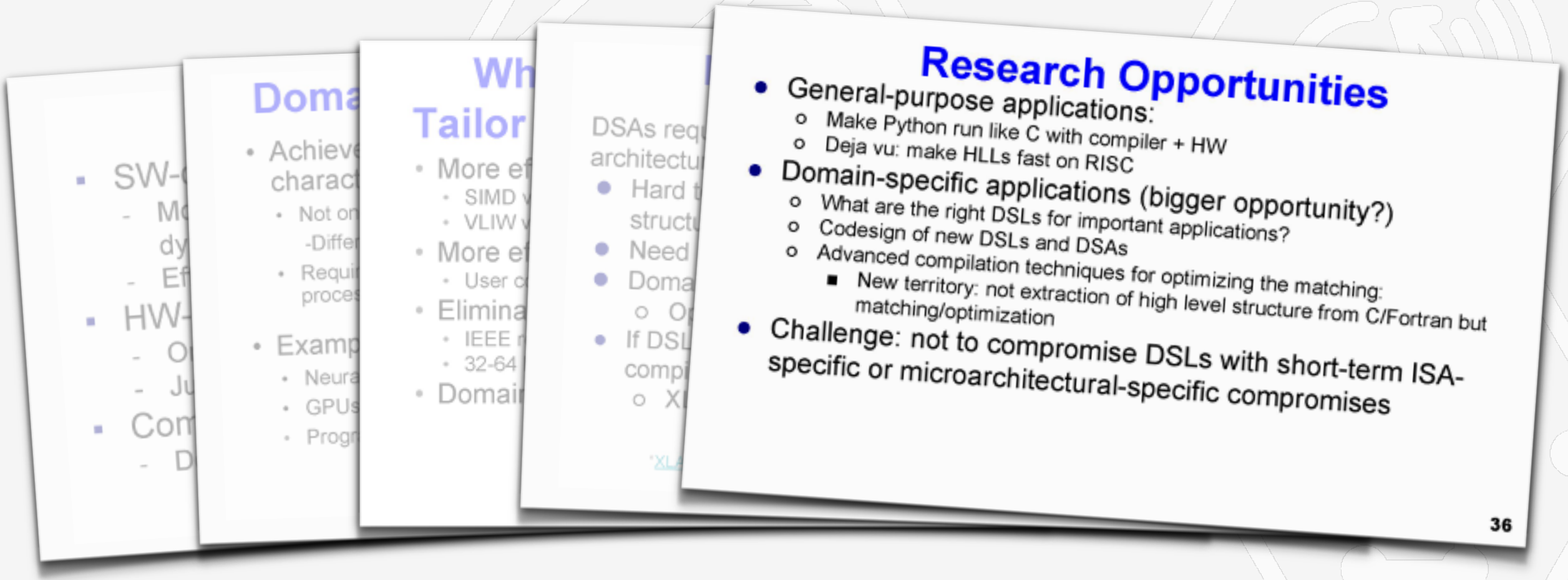


Domain Specific Languages for Domain Specific Architectures

This material is based on research sponsored by the Defense Advanced Research Projects Agency (DARPA) under agreement HR0011-18-3-0007; the Application Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA; the Toyota Research Institute; the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award Numbers DE-SC0008923 and DE-SC0018121; and the National Science Foundation under Grant No. CCF-1533753;

“A GOLDEN AGE FOR COMPUTER ARCHITECTURE”

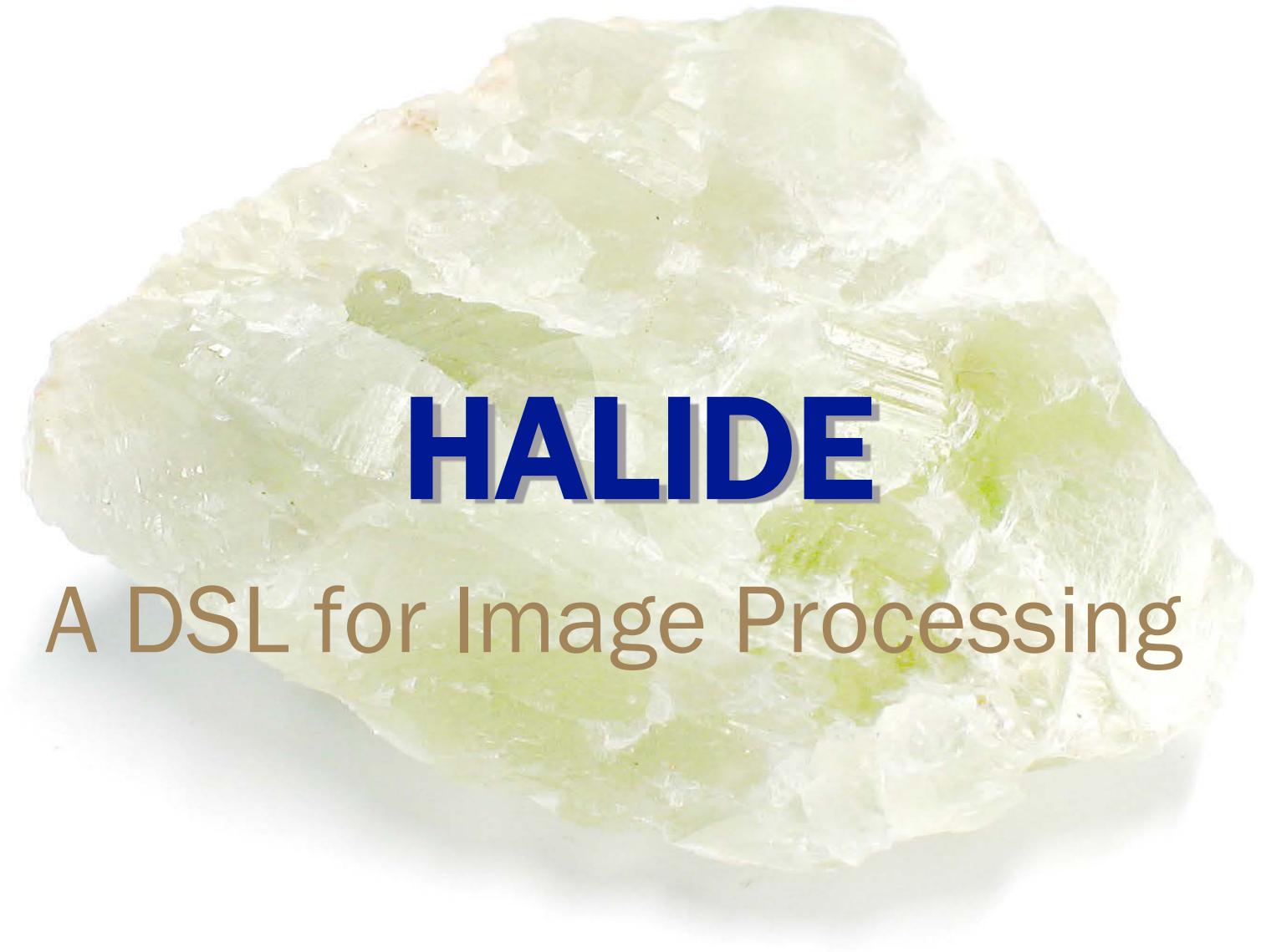
TURING AWARD LECTURE BY HENNESSY AND PATTERSON, 2018



From the Turing Award Lecture by Hennessy and Patterson

WHAT IS A DOMAIN SPECIFIC LANGUAGE?

- A Domain Specific Language capture expert knowledge about an application domain
- DSLs provide a high-level programming model for domain scientists
  - Capture the programmer intent at a higher level of abstraction
- DSLs provide the compiler more opportunities for optimization
  - Can encode domain expert knowledge of specific optimizations
  - Better view of the intended computation without heroic analysis
  - Less low-level decisions by the programmer that has to be undone
  - Able to generate high-performance code for multiple hardware platforms



HALIDE

A DSL for Image Processing

Image from the Smithsonian
National Museum of Natural History

A SIMPLE EXAMPLE: 3X3 BLUR

```
void box_filter_3x3(const Image &in, Image &blury) {

  Image blurx(in.width(), in.height());  // allocate blurx array


  for (int y = 0; y < in.height(); y++)
    for (int x = 0; x < in.height(); x++)
      blurx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;


  for (int x = 0; x < in.width(); x++)
    for (int y = 0; y < in.height(); y++)
      blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;

}
```

HAND-OPTIMIZED C++

```
void box_filter_3x3(const Image &in, Image &blury) {
  __m128i one_third = _mm_set1_epi16(21846);
  #pragma omp parallel for
  for (int yTile = 0; yTile < in.height(); yTile += 32) {
    __m128i a, b, c, sum, avg;
    __m128i blurx[(256/8)*(32+2)]; // allocate tile blurx array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
      __m128i *blurxPtr = blurx;
      for (int y = -1; y < 32+1; y++) {
        const uint16_t *inPtr = &(in[yTile+y][xTile]);
        for (int x = 0; x < 256; x += 8) {
         a = _mm_loadu_si128((__m128i*)(inPtr-1));
         b = _mm_loadu_si128((__m128i*)(inPtr+1));
         c = _mm_load_si128((__m128i*)(inPtr));
         sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
         avg = _mm_mulhi_epi16(sum, one_third);
         _mm_store_si128(blurxPtr++, avg);
         inPtr += 8;
      }}
      blurxPtr = blurx;
      for (int y = 0; y < 32; y++) {
        __m128i *outPtr = (__m128i *)(&(blury[yTile+y][xTile]));
        for (int x = 0; x < 256; x += 8) {
          a = _mm_load_si128(blurxPtr+(2*256)/8);
          b = _mm_load_si128(blurxPtr+256/8);
          c = _mm_load_si128(blurxPtr++);
          sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(outPtr++, avg);
}}}}}
```

9.9 → 0.9 ms/megapixel

11x faster
(quad core x86)

Tiled, fused

Vectorized

Multithreaded

Redundant computation

Near roof-line optimum

HALIDE DECOUPLES ALGORITHM FROM SCHEDULE

- Algorithm: *what* is computed
  - The algorithm defines pipelines as pure functions
  - Pipeline stages are functions from coordinates to values
  - Execution order and storage are unspecified

```
blurx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;
```

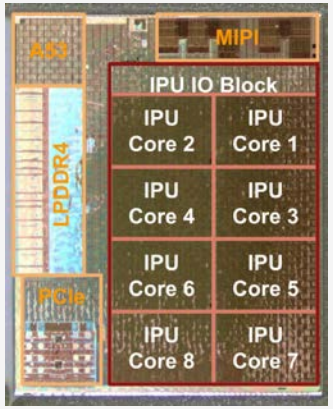
HALIDE DECOUPLES ALGORITHM FROM SCHEDULE

- Algorithm: *what* is computed
  - The algorithm defines pipelines as pure functions
  - Pipeline stages are functions from coordinates to values
  - Execution order and storage are unspecified

- Schedule: *where* and *when* it's computed
  - Architecture Specific
  - Single, unified model for all schedules
  - Simple enough to search, expose to user
    Powerful enough to beat expert-tuned code

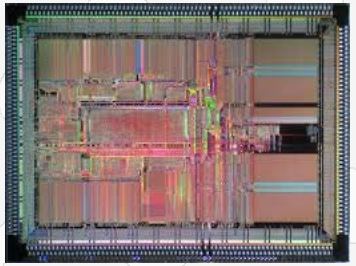
DSAS FOR IMAGE PROCESSING

Google Pixel Visual Core



Qualcomm Snapdragon

Hexagon 682 DSP



- Halide is the DSL for programming these DSAs
- Programmers able to find the best hardware utilization using the scheduling language
- Thus, the DSAs are more general purpose and easy to use

Image from Anand Tech and commons wikimedia

TACO

A DSL for Sparse Tensor Algebra

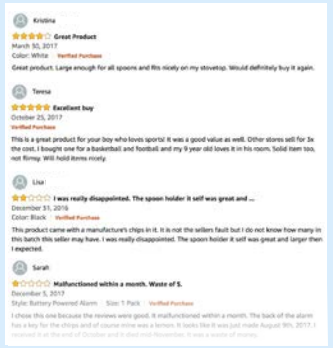
Image from the Wikimedia commons

TENSORS ARE EVERYWHERE

Data Analytics

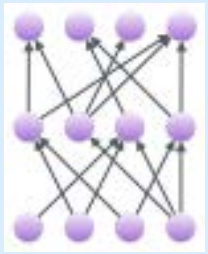
Movies

Social Networks

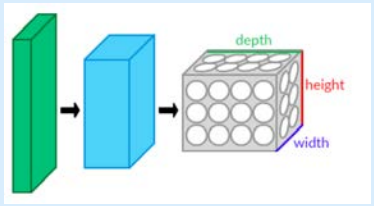


Product Reviews

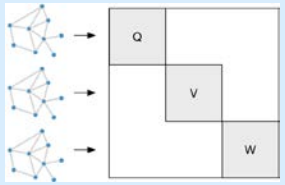
Machine Learning



Sparse Networks

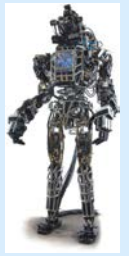


Sparse Convolutional Networks



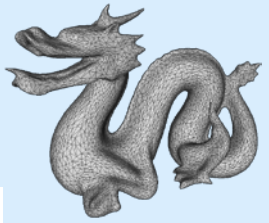
Graph Convolutional Network

Science and Engineering



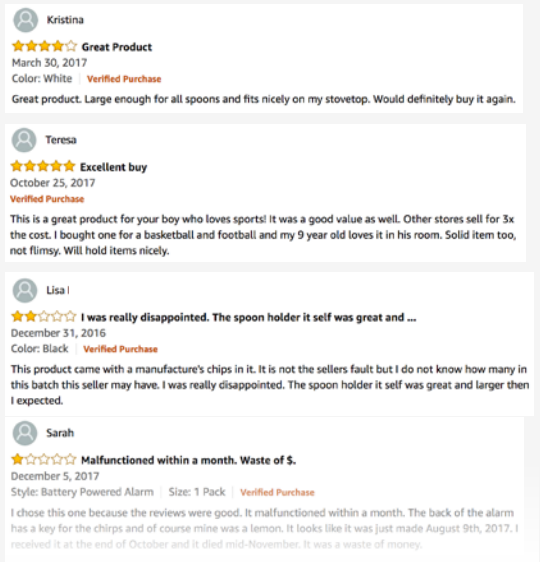
Robotics

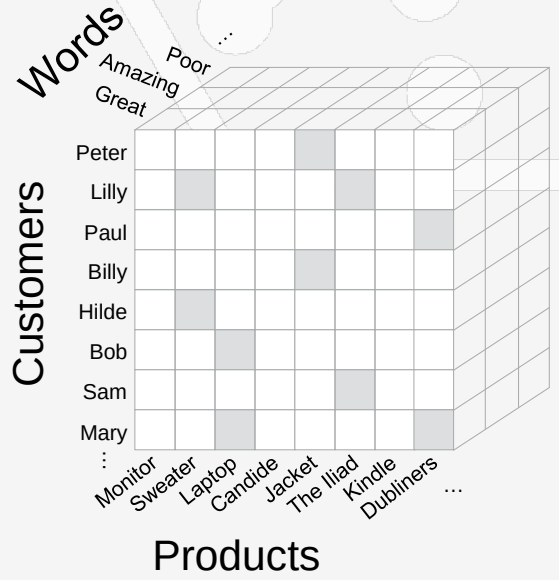
Computational Biology



Simulations

Amazon Product Reviews





Extremely sparse

Dense storage: 107 Exabytes

Sparse storage: 13 Gigabytes

Images from the Wikimedia and computational Imaging

COMPLEXITY OF SPARSE CODE

$$A_{ijk} = B_{ijk} + C_{ijk}$$

CSF COO

```
int iB = 0;
int C0_pos = C0_pos[0];
while (C0_pos < C0_pos[1]) {
  int iC = C0_crd[C0_pos];
  int C0_end = C0_pos + 1;
  if (iC == iB)
    while ((C0_end < C0_pos[1]) && (C0_crd[C0_end] == iB)) {
      C0_end++;
    }
  if (iC == iB) {
    int B1_pos = B1_pos[iB];
    int C1_pos = C0_pos;
    while ((B1_pos < B1_pos[iB + 1]) && (C1_pos < C0_end)) {
      int jB = B1_crd[B1_pos];
      int jC = C1_crd[C1_pos];
      int j = min(jB, jC);
      int A1_pos = (iB * A1_size) + j;
      int C1_end = C1_pos + 1;
      if (jC == j)
        while ((C1_end < C0_end) && (C1_crd[C1_end] == j)) {
          C1_end++;
        }
      if ((jB == j) && (jC == j)) {
        int B2_pos = B2_pos[B1_pos];
        int C2_pos = C1_pos;
        while ((B2_pos < B2_pos[B1_pos + 1]) && (C2_pos < C1_end)) {
          int kB = B2_crd[B2_pos];
          int kC = C2_crd[C2_pos];
          int k = min(kB, kC);
          int A2_pos = (A1_pos * A2_size) + k;
          if ((kB == k) && (kC == k)) {
            A[A2_pos] = B[B2_pos] + C[C2_pos];
          } else if (kB == k) {
            A[A2_pos] = B[B2_pos];
          } else {
            A[A2_pos] = C[C2_pos];
          }
          if (kB == k) B2_pos++;
          if (kC == k) C2_pos++;
        }
        while (B2_pos < B2_pos[B1_pos + 1]) {
          int kB0 = B2_crd[B2_pos];
          int A2_pos0 = (A1_pos * A2_size) + kB0;
          A[A2_pos0] = B[B2_pos];
          B2_pos++;
        }
        while (C2_pos < C1_end) {
          int kC0 = C2_crd[C2_pos];
          int A2_pos1 = (A1_pos * A2_size) + kC0;
          A[A2_pos1] = C[C2_pos];
          C2_pos++;
        }
      } else if (jB == j) {
        for (int B2_pos0 = B2_pos[B1_pos];
             B2_pos0 < B2_pos[B1_pos + 1]; B2_pos0++) {
          int kB1 = B2_crd[B2_pos0];
          int A2_pos2 = (A1_pos * A2_size) + kB1;
          A[A2_pos2] = B[B2_pos0];
        }
      } else {
        for (int C2_pos0 = C1_pos; C2_pos0 < C1_end; C2_pos0++) {
          int kC1 = C2_crd[C2_pos0];
          int A2_pos3 = (A1_pos * A2_size) + kC1;
          A[A2_pos3] = C[C2_pos0];
        }
      }
      if (jB == j) B1_pos++;
      if (jC == j) C1_pos = C1_end;
    }
    while (B1_pos < B1_pos[iB + 1]) {
      int jB0 = B1_crd[B1_pos];
      int A1_pos0 = (iB * A1_size) + jB0;
      for (int B2_pos1 = B2_pos[B1_pos];
           B2_pos1 < B2_pos[B1_pos + 1]; B2_pos1++) {
        int kB2 = B2_crd[B2_pos1];
        int A2_pos4 = (A1_pos0 * A2_size) + kB2;
        A[A2_pos4] = B[B2_pos1];
      }
      B1_pos++;
    }
    while (C1_pos < C0_end) {
      int jC0 = C1_crd[C1_pos];
      int A1_pos1 = (iB * A1_size) + jC0;
      int C1_end0 = C1_pos + 1;
      while ((C1_end0 < C0_end) && (C1_crd[C1_end0] == jC0)) {
        C1_end0++;
      }
      for (int C2_pos1 = C1_pos; C2_pos1 < C1_end0; C2_pos1++) {
        int kC2 = C2_crd[C2_pos1];
        int A2_pos5 = (A1_pos1 * A2_size) + kC2;
        A[A2_pos5] = C[C2_pos1];
      }
      C1_pos = C1_end0;
    }
  } else {
    for (int B1_pos0 = B1_pos[iB];
         B1_pos0 < B1_pos[iB + 1]; B1_pos0++) {
      int jB1 = B1_crd[B1_pos0];
      int A1_pos2 = (iB * A1_size) + jB1;
      for (int B2_pos2 = B2_pos[B1_pos0];
           B2_pos2 < B2_pos[B1_pos0 + 1]; B2_pos2++) {
        int kB3 = B2_crd[B2_pos2];
        int A2_pos6 = (A1_pos2 * A2_size) + kB3;
        A[A2_pos6] = B[B2_pos2];
      }
    }
  }
  if (iC == iB) C0_pos = C0_end;
  iB++;
}
while (iB < B0_size) {
  for (int B1_pos1 = B1_pos[iB];
       B1_pos1 < B1_pos[iB + 1]; B1_pos1++) {
    int jB2 = B1_crd[B1_pos1];
    int A1_pos3 = (iB * A1_size) + jB2;
    for (int B2_pos3 = B2_pos[B1_pos1];
         B2_pos3 < B2_pos[B1_pos1 + 1]; B2_pos3++) {
      int kB4 = B2_crd[B2_pos3];
      int A2_pos7 = (A1_pos3 * A2_size) + kB4;
      A[A2_pos7] = B[B2_pos3];
    }
  }
  iB++;
}
```

TOO MANY TENSOR KERNELS FOR A FIXED-FUNCTION LIBRARY

CSparse

Eigen (SpMV)

$$a = Bc \qquad a = Bc + a \qquad a = Bc + b \qquad A = B + C \qquad a = \alpha Bc + \beta a$$

OSKI

OSKI has 282 specialized variants of this expression

PETSc

$$a = B^T c \qquad A = \alpha B \qquad a = B(c + d) \qquad a = B^T c + d \qquad A = B + C + D \qquad A = BC$$

$$A = B \odot C \qquad a = b \odot c \qquad A = 0 \qquad A = B \odot (CD) \qquad A = BCd \qquad A = B^T \qquad a = B^T Bc$$

$$a = b + c \qquad A = B \qquad K = A^T C A$$

$$A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj} \qquad A_{kj} = \sum_{il} B_{ikl} C_{lj} D_{ij}$$

$$A_{lj} = \sum_{ik} B_{ikl} C_{ij} D_{kj} \qquad A_{ij} = \sum_{k} B_{ijk} c_k$$

$$A_{ijk} = \sum_{l} B_{ikl} C_{lj} \qquad A_{ik} = \sum_{j} B_{ijk} c_j$$

$$A_{jk} = \sum_{i} B_{ijk} c_i \qquad A_{ijl} = \sum_{k} B_{ikl} C_{kj}$$

$$C = \sum_{ijkl} M_{ij} P_{jk} \overline{M_{lk}}\, \overline{P_{il}} \qquad \tau = \sum_{i} z_i \left(\sum_{j} z_j \theta_{ij}\right)\left(\sum_{k} z_k \theta_{ik}\right)$$

$$a = \sum_{ijklmnop} M_{ij} P_{jk} M_{kl} P_{lm} \overline{M_{nm}} P_{no} \overline{M_{po}}\, \overline{P_{ip}}$$

Linear Algebra

Dense Matrix

CSR DCSR BCSR

COO ELLPACK CSB

Blocked COO CSC

DIA Blocked DIA DCSC

Sparse vector Hash Maps

Coordinates

CSF Dense Tensors

Blocked Tensors

Linked Lists Database

Compression Schemes

Cloud Storage

Data analytics
(tensor factorization)

Quantum Chromodynamics

Finite Elements Method,
Block-Sparse NN Weights [GRK 2017]

Data Analytics

Convolutions, Image Processing

CPU

GPUs

TPUs

FPGA

Sparse Tensor Hardware

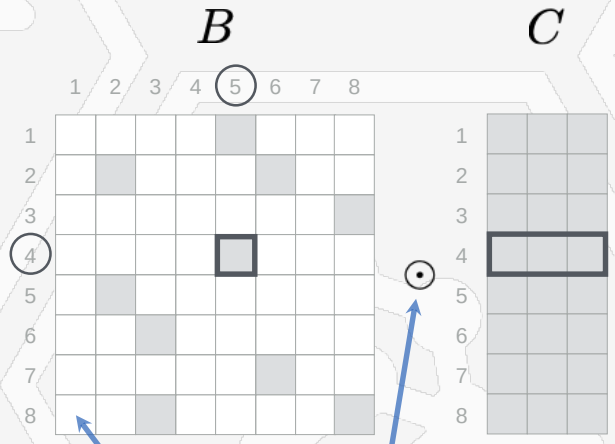
Cloud Computers

Supercomputers

MANY NON-BINARY EXPRESSIONS MUST BE COMPUTED IN A SINGLE KERNEL

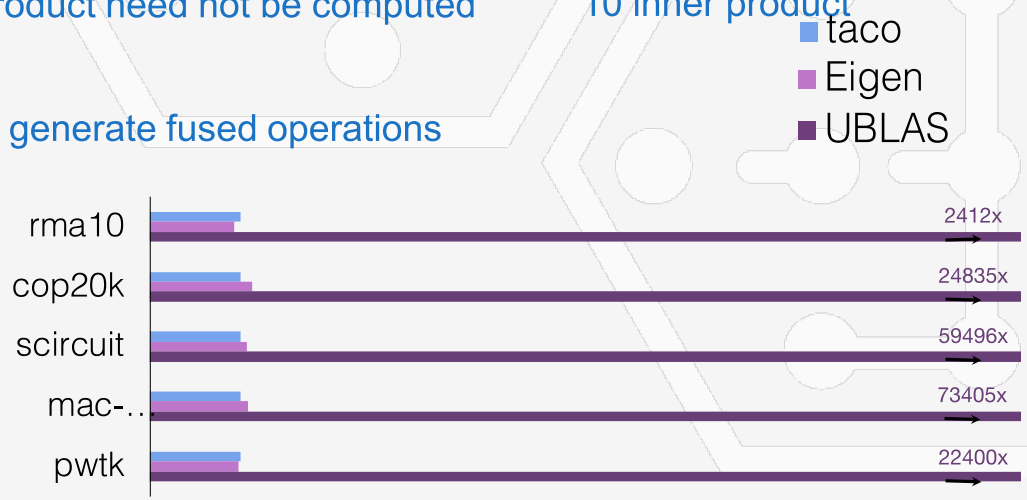
$$a = Bc \quad a = Bc + a \quad a = Bc + b \quad A = B + C \quad a = \alpha Bc + \beta a$$
$$a = B^T c \quad A = \alpha B \quad a = B(c + d)$$
$$a = B^T c + d \quad A = B + C + D \quad A = BC$$
$$A = B \odot C \quad a = b \odot c \quad A = 0 \quad \mathbf{A = B \odot (CD)}$$
$$A = BCd \quad A = B^T \quad a = B^T Bc$$
$$a = b + c \quad A = B \quad K = A^T C A$$
$$A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj} \quad A_{kj} = \sum_{il} B_{ikl} C_{lj} D_{ij}$$
$$A_{lj} = \sum_{ik} B_{ikl} C_{ij} D_{kj} \quad A_{ij} = \sum_{k} B_{ijk} c_k$$
$$A_{ijk} = \sum_{l} B_{ikl} C_{lj} \quad A_{ik} = \sum_{j} B_{ijk} c_j$$
$$A_{jk} = \sum_{i} B_{ijk} c_i \quad A_{ijl} = \sum_{k} B_{ikl} C_{kj}$$
$$\tau = \sum_i z_i (\sum_j z_j \theta_{ij})(\sum_k z_k \theta_{ik})$$
$$C = \sum_{ijkl} M_{ij} P_{jk} \overline{M_{lk}}\, \overline{P_{il}}$$
$$a = \sum_{ijklmnop} M_{ij} P_{jk} M_{kl} P_{lm} \overline{M_{nm}} P_{no} \overline{M_{po}}\, \overline{P_{ip}}$$

Sampled Dense-Dense Matrix Multiplication (SDDMM)



~~64 inner product~~
10 inner product

We will generate fused operations



MANY NON-BINARY EXPRESSIONS MUST BE COMPUTED IN A SINGLE KERNEL

$$a = Bc$$

$$a = Bc + a \quad a = Bc + b \quad A = B + C \quad a = \alpha Bc + \beta a \quad a = B^T c \quad A = \alpha B \quad a = B(c + d)$$

$$a = B^T c + d \quad A = B + C + D \quad A = BC \quad A = B \odot C \quad a = b \odot c A = 0$$

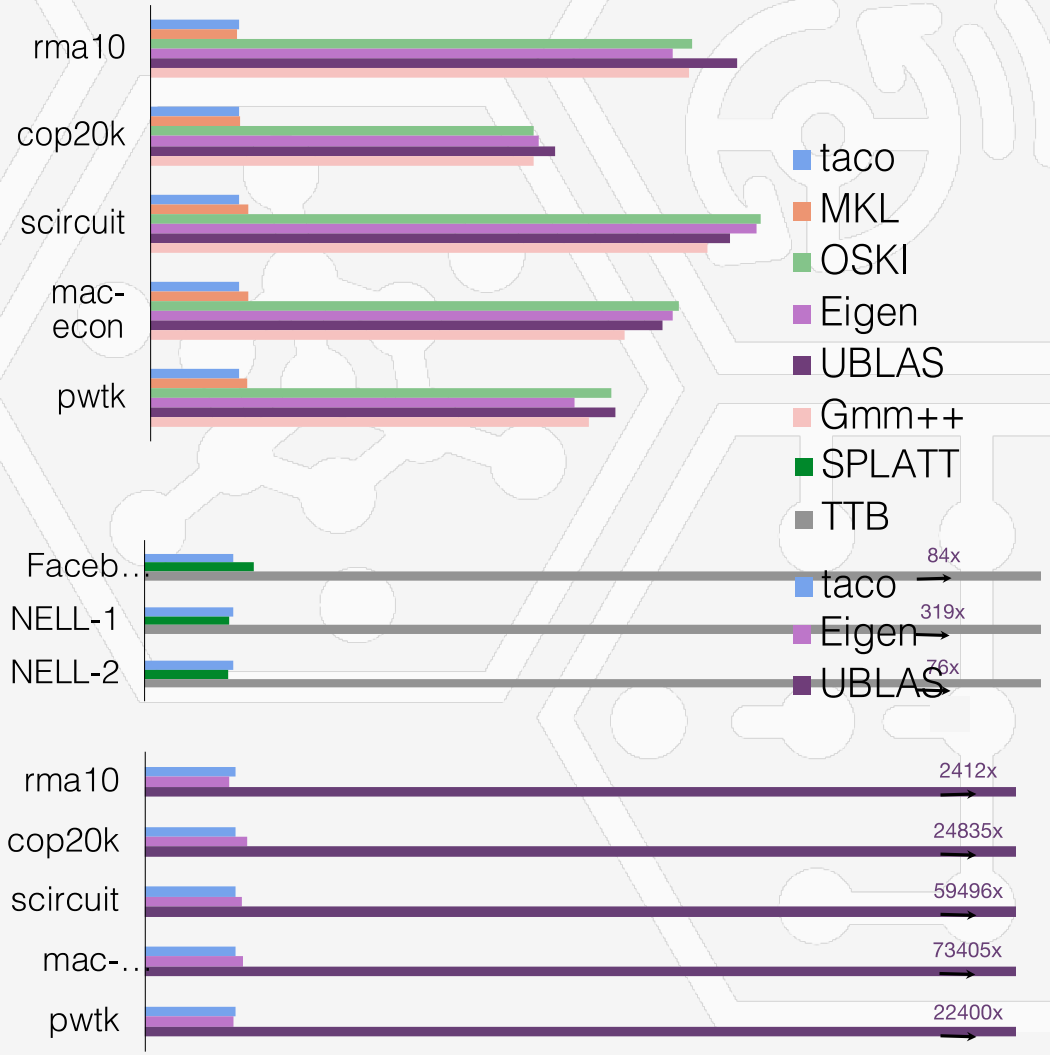
$$A = B \odot (CD) \quad A = BCd \quad A = B^T \quad a = B^T Bc \quad a = b + c \quad A = B \quad K = A^T C A$$

$$A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj} \quad A_{kj} = \sum_{il} B_{ikl} C_{lj} D_{ij} \quad A_{lj} = \sum_{ik} B_{ikl} C_{ij} D_{kj} \quad A_{ij} = \sum_{k} B_{ijk} c_k$$

$$A_{ijk} = \sum_{l} B_{ikl} C_{lj} \quad A_{ik} = \sum_{j} B_{ijk} c_j \quad A_{jk} = \sum_{i} B_{ijk} c_i \quad A_{ijl} = \sum_{k} B_{ikl} C_{kj}$$

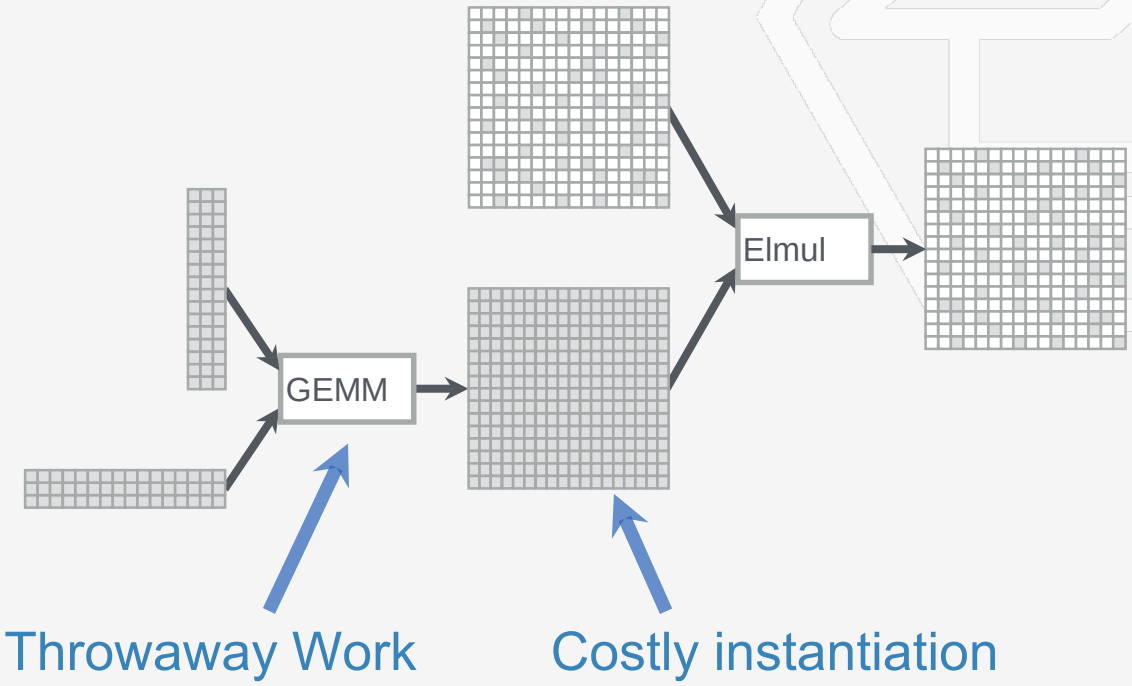
$$\tau = \sum_i z_i (\sum_j z_j \theta_{ij})(\sum_k z_k \theta_{ik}) \quad C = \sum_{ijkl} M_{ij} P_{jk} \overline{M_{lk}}\, \overline{P_{il}}$$

$$a = \sum_{ijklmnop} M_{ij} P_{jk} M_{kl} P_{lm} \overline{M_{nm}} P_{no} \overline{M_{po}}\, \overline{P_{ip}}$$



PERFORMANCE COST OF LIBRARY ABSTRACTION IS HIGH

Fixed-function Libraries



Hand-Optimized Code

```
int iB = 0;
int C0_pos = C0_pos_arr[0];
while (C0_pos < C0_pos_arr[1]) {
  int iC = C0_idx_arr[C0_pos];
  int C0_end = C0_pos + 1;
  if (iC == iB)
    while ((C0_end < C0_pos_arr[1]) && (C0_idx_arr[C0_end] == iB)) {
      C0_end++;
    }
  if (iC == iB) {
    int B1_pos = B1_pos_arr[iB];
    int C1_pos = C0_pos;
    while ((B1_pos < B1_pos_arr[iB + 1]) && (C1_pos < C0_end)) {
      int jB = B1_idx_arr[B1_pos];
      int jC = C1_idx_arr[C1_pos];
      int j = min(jB, jC);
      int A1_pos = (iB * A1_size) + j;
      int C1_end = C1_pos + 1;
      if (jC == j)
        while ((C1_end < C0_end) && (C1_idx_arr[C1_end] == j)) {
          C1_end++;
        }
      if ((jB == j) && (jC == j)) {
        int B2_pos = B2_pos_arr[B1_pos];
        int C2_pos = C1_pos;
        while ((B2_pos < B2_pos_arr[B1_pos + 1]) && (C2_pos < C1_end)) {
          int kB = B2_idx_arr[B2_pos];
          int kC = C2_idx_arr[C2_pos];
          int k = min(kB, kC);
          int A2_pos = (A1_pos * A2_size) + k;
          if ((kB == k) && (kC == k)) {
            A_val_arr[A2_pos] = B_val_arr[B2_pos] + C_val_arr[C2_pos];
          } else if (kB == k) {
            A_val_arr[A2_pos] = B_val_arr[B2_pos];
          } else {
            A_val_arr[A2_pos] = C_val_arr[C2_pos];
          }
          if (kB == k) B2_pos++;
          if (kC == k) C2_pos++;
        }
        while (B2_pos < B2_pos_arr[B1_pos + 1]) {
          int kB0 = B2_idx_arr[B2_pos];
          int A2_pos0 = (A1_pos * A2_size) + kB0;
          A_val_arr[A2_pos0] = B_val_arr[B2_pos];
          B2_pos++;
        }
        while (C2_pos < C1_end) {
          int kC0 = C2_idx_arr[C2_pos];
          int A2_pos1 = (A1_pos * A2_size) + kC0;
          A_val_arr[A2_pos1] = C_val_arr[C2_pos];
          C2_pos++;
        }
      } else if (jB == j) {
        for (int B2_pos0 = B2_pos_arr[B1_pos];
             B2_pos0 < B2_pos_arr[B1_pos + 1]; B2_pos0++) {
          int kB1 = B2_idx_arr[B2_pos0];
          int A2_pos2 = (A1_pos * A2_size) + kB1;
          A_val_arr[A2_pos2] = B_val_arr[B2_pos0];
        }
      } else {
        for (int C2_pos0 = C1_pos; C2_pos0 < C1_end; C2_pos0++) {
          int kC1 = C2_idx_arr[C2_pos0];
          int A2_pos3 = (A1_pos * A2_size) + kC1;
          A_val_arr[A2_pos3] = C_val_arr[C2_pos0];
        }
      }
      if (jB == j) B1_pos++;
      if (jC == j) C1_pos = C1_end;
    }
```

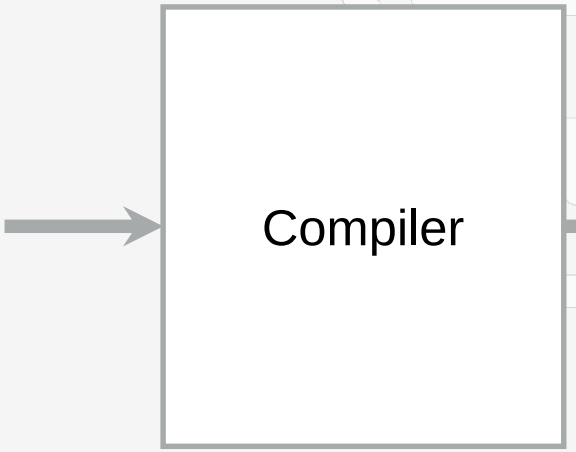
```
    while (B1_pos < B1_pos_arr[iB + 1]) {
      int jB0 = B1_idx_arr[B1_pos];
      int A1_pos0 = (iB * A1_size) + jB0;
      for (int B2_pos1 = B2_pos_arr[B1_pos];
               B2_pos1 < B2_pos_arr[B1_pos + 1]; B2_pos1++) {
        int kB2 = B2_idx_arr[B2_pos1];
        int A2_pos4 = (A1_pos0 * A2_size) + kB2;
        A_val_arr[A2_pos4] = B_val_arr[B2_pos1];
      }
      B1_pos++;
    }
    while (C1_pos < C0_end) {
      int jC0 = C1_idx_arr[C1_pos];
      int A1_pos1 = (iB * A1_size) + jC0;
      int C1_end0 = C1_pos + 1;
      while ((C1_end0 < C0_end) && (C1_idx_arr[C1_end0] == jC0)) {
        C1_end0++;
      }
      for (int C2_pos1 = C1_pos; C2_pos1 < C1_end0; C2_pos1++) {
        int kC2 = C2_idx_arr[C2_pos1];
        int A2_pos5 = (A1_pos1 * A2_size) + kC2;
        A_val_arr[A2_pos5] = C_val_arr[C2_pos1];
      }
      C1_pos = C1_end0;
    }
  } else {
    for (int B1_pos0 = B1_pos_arr[iB];
             B1_pos0 < B1_pos_arr[iB + 1]; B1_pos0++) {
      int jB1 = B1_idx_arr[B1_pos0];
      int A1_pos2 = (iB * A1_size) + jB1;
      for (int B2_pos2 = B2_pos_arr[B1_pos0];
               B2_pos2 < B2_pos_arr[B1_pos0 + 1]; B2_pos2++) {
        int kB3 = B2_idx_arr[B2_pos2];
        int A2_pos6 = (A1_pos2 * A2_size) + kB3;
        A_val_arr[A2_pos6] = B_val_arr[B2_pos2];
      }
    }
  }
  if (iC == iB) C0_pos = C0_end;
  iB++;
}
while (iB < B0_size) {
  for (int B1_pos1 = B1_pos_arr[iB];
           B1_pos1 < B1_pos_arr[iB + 1]; B1_pos1++) {
    int jB2 = B1_idx_arr[B1_pos1];
    int A1_pos3 = (iB * A1_size) + jB2;
    for (int B2_pos3 = B2_pos_arr[B1_pos1];
             B2_pos3 < B2_pos_arr[B1_pos1 + 1]; B2_pos3++) {
      int kB4 = B2_idx_arr[B2_pos3];
      int A2_pos7 = (A1_pos3 * A2_size) + kB4;
      A_val_arr[A2_pos7] = B_val_arr[B2_pos3];
    }
  }
  iB++;
}
```

$$A_{ijk} = B_{ijk} + C_{ijk}$$

ABSTRACTION WITHOUT FRICTION

Language and Compiler Abstractions

Domain Specific Languages



Generated Code

```
int iB = 0;
int C0_pos = C0_pos_arr[0];
while (C0_pos < C0_pos_arr[1]) {
  int iC = C0_idx_arr[C0_pos];
  int C0_end = C0_pos + 1;
  if (iC == iB)
    while ((C0_end < C0_pos_arr[1]) && (C0_idx_arr[C0_end] == iB)) {
      C0_end++;
    }
  if (iC == iB) {
    int B1_pos = B1_pos_arr[iB];
    int C1_pos = C0_pos;
    while ((B1_pos < B1_pos_arr[iB + 1]) && (C1_pos < C0_end)) {
      int jB = B1_idx_arr[B1_pos];
      int jC = C1_idx_arr[C1_pos];
      int j = min(jB, jC);
      int A1_pos = (iB * A1_size) + j;
      int C1_end = C1_pos + 1;
      if (jC == j)
        while ((C1_end < C0_end) && (C1_idx_arr[C1_end] == j)) {
          C1_end++;
        }
      if ((jB == j) && (jC == j)) {
        int B2_pos = B2_pos_arr[B1_pos];
        int C2_pos = C1_pos;
        while ((B2_pos < B2_pos_arr[B1_pos + 1]) && (C2_pos < C1_end)) {
          int kB = B2_idx_arr[B2_pos];
          int kC = C2_idx_arr[C2_pos];
          int k = min(kB, kC);
          int A2_pos = (A1_pos * A2_size) + k;
          if ((kB == k) && (kC == k)) {
            A_val_arr[A2_pos] = B_val_arr[B2_pos] + C_val_arr[C2_pos];
          } else if (kB == k) {
            A_val_arr[A2_pos] = B_val_arr[B2_pos];
          } else {
            A_val_arr[A2_pos] = C_val_arr[C2_pos];
          }
          if (kB == k) B2_pos++;
          if (kC == k) C2_pos++;
        }
        while (B2_pos < B2_pos_arr[B1_pos + 1]) {
          int kB0 = B2_idx_arr[B2_pos];
          int A2_pos0 = (A1_pos * A2_size) + kB0;
          A_val_arr[A2_pos0] = B_val_arr[B2_pos];
          B2_pos++;
        }
        while (C2_pos < C1_end) {
          int kC0 = C2_idx_arr[C2_pos];
          int A2_pos1 = (A1_pos * A2_size) + kC0;
          A_val_arr[A2_pos1] = C_val_arr[C2_pos];
          C2_pos++;
        }
      } else if (jB == j) {
        for (int B2_pos0 = B2_pos_arr[B1_pos];
             B2_pos0 < B2_pos_arr[B1_pos + 1]; B2_pos0++) {
          int kB1 = B2_idx_arr[B2_pos0];
          int A2_pos2 = (A1_pos * A2_size) + kB1;
          A_val_arr[A2_pos2] = B_val_arr[B2_pos0];
        }
      } else {
        for (int C2_pos0 = C1_pos; C2_pos0 < C1_end; C2_pos0++) {
          int kC1 = C2_idx_arr[C2_pos0];
          int A2_pos3 = (A1_pos * A2_size) + kC1;
          A_val_arr[A2_pos3] = C_val_arr[C2_pos0];
        }
      }
      if (jB == j) B1_pos++;
      if (jC == j) C1_pos = C1_end;
    }
    while (B1_pos < B1_pos_arr[iB + 1]) {
      int jB0 = B1_idx_arr[B1_pos];
      int A1_pos0 = (iB * A1_size) + jB0;
      for (int B2_pos1 = B2_pos_arr[B1_pos];
           B2_pos1 < B2_pos_arr[B1_pos + 1]; B2_pos1++) {
        int kB2 = B2_idx_arr[B2_pos1];
        int A2_pos4 = (A1_pos0 * A2_size) + kB2;
        A_val_arr[A2_pos4] = B_val_arr[B2_pos1];
      }
      B1_pos++;
    }
    while (C1_pos < C0_end) {
      int jC0 = C1_idx_arr[C1_pos];
      int A1_pos1 = (iB * A1_size) + jC0;
      int C1_end0 = C1_pos + 1;
      while ((C1_end0 < C0_end) && (C1_idx_arr[C1_end0] == jC0)) {
        C1_end0++;
      }
      for (int C2_pos1 = C1_pos; C2_pos1 < C1_end0; C2_pos1++) {
        int kC2 = C2_idx_arr[C2_pos1];
        int A2_pos5 = (A1_pos1 * A2_size) + kC2;
        A_val_arr[A2_pos5] = C_val_arr[C2_pos1];
      }
      C1_pos = C1_end0;
    }
  } else {
    for (int B1_pos0 = B1_pos_arr[iB];
         B1_pos0 < B1_pos_arr[iB + 1]; B1_pos0++) {
      int jB1 = B1_idx_arr[B1_pos0];
      int A1_pos2 = (iB * A1_size) + jB1;
      for (int B2_pos2 = B2_pos_arr[B1_pos0];
           B2_pos2 < B2_pos_arr[B1_pos0 + 1]; B2_pos2++) {
        int kB3 = B2_idx_arr[B2_pos2];
        int A2_pos6 = (A1_pos2 * A2_size) + kB3;
        A_val_arr[A2_pos6] = B_val_arr[B2_pos2];
      }
    }
  }
  if (iC == iB) C0_pos = C0_end;
  iB++;
}
while (iB < B0_size) {
  for (int B1_pos1 = B1_pos_arr[iB];
       B1_pos1 < B1_pos_arr[iB + 1]; B1_pos1++) {
    int jB2 = B1_idx_arr[B1_pos1];
    int A1_pos3 = (iB * A1_size) + jB2;
    for (int B2_pos3 = B2_pos_arr[B1_pos1];
         B2_pos3 < B2_pos_arr[B1_pos1 + 1]; B2_pos3++) {
      int kB4 = B2_idx_arr[B2_pos3];
      int A2_pos7 = (A1_pos3 * A2_size) + kB4;
      A_val_arr[A2_pos7] = B_val_arr[B2_pos3];
    }
  }
  iB++;
}
```

$$A_{ijk} = B_{ijk} + C_{ijk}$$

THE TENSOR ALGEBRA COMPILER (TACO)

Expressions

$$A = Bc + a \qquad a = Bc$$
$$A = B \odot C \qquad A = B + C \qquad a = \alpha Bc + \beta a$$
$$A = BCd \qquad A = \alpha B \quad A = 0 \quad A = BC$$
$$a = b \odot c \qquad A = B \odot (CD)$$
$$A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj} \qquad A = B^T \quad a = B^T Bc$$
$$A_{ik} = \sum_{j} B_{ijk} c_j \quad A_{kj} = \sum_{il} B_{ikl} C_{lj} D_{ij}$$
$$A_{ijk} = \sum_{l} B_{ikl} C_{lj} \qquad A_{ij} = (\sum_{k} B_{ijk} C_{ijk}) + D_{ij}$$
$$C = \sum_{ijkl} M_{ij} P_{jk} \overline{M_{lk}} \overline{P_{il}} \qquad \tau = \sum_{i} z_i (\sum_{j} z_j \theta_{ij}) (\sum_{k} z_k \theta_{ik})$$
$$a = \sum_{ijklmnop} M_{ij} P_{jk} M_{kl} P_{lm} \overline{M_{nm}} P_{no} \overline{M_{po}} \overline{P_{ip}}$$

Formats

Dense Matrix, CSR, BCSR, DCSR, ELLPACK, COO, CSB, DIA, Blocked COO, CSC, Blocked DIA, DCSC, Hash Maps, Sparse vector, Dense Tensors, CSF, Blocked Tensors

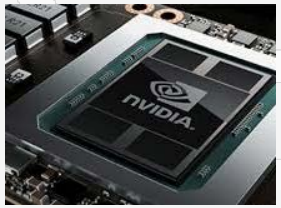
Tensor Algebra Compiler (taco)

THE C PROGRAMMING LANGUAGE

NVIDIA CUDA

Legion/MPI

DARPA SDH
Nvidia Symphony DSA

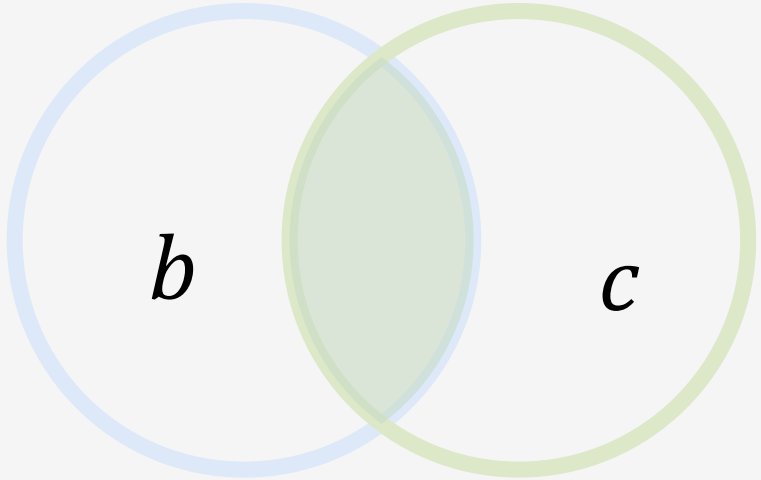


Images from the Wikimedia commons

AVOID WASTED WORK AND ITERATIONS

$$a_i = b_i c_i$$

Multiplication requires intersection

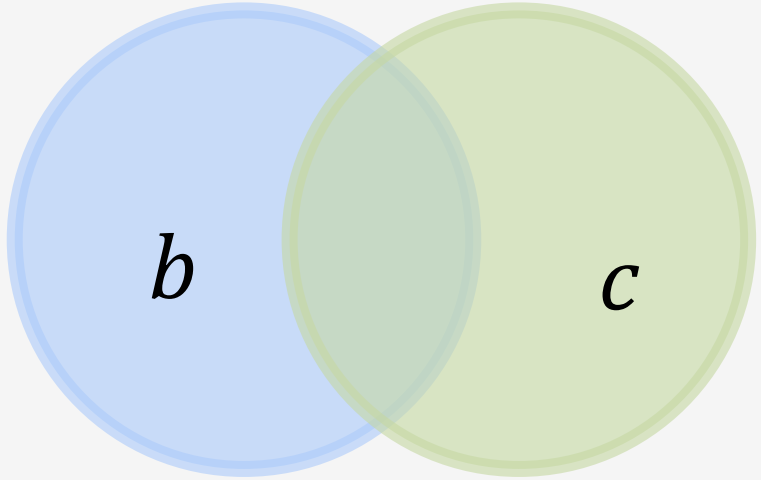



```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
  int ib = b1_crd[pb1];
  int ic = c1_crd[pc1];
  int i = min(ib, ic);
  if (ib == i && ic == i) {
    a[i] = b[pb1] * c[pc1];
  }
  if (ib == i) pb1++;
  if (ic == i) pc1++;
}
```

AVOID WASTED WORK AND ITERATIONS

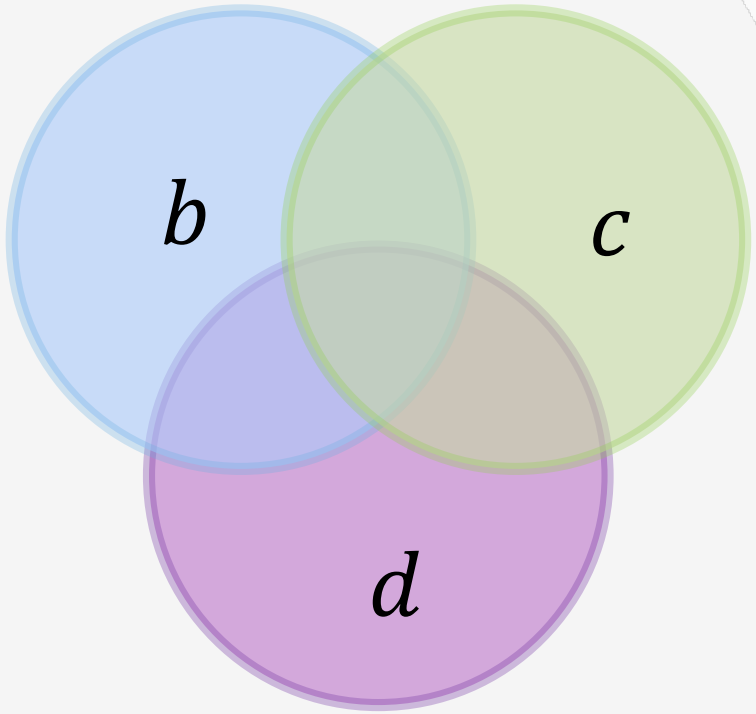
$$a_i = b_i + c_i$$

Addition requires union




```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
  int ib = b1_crd[pb1];
  int ic = c1_crd[pc1];
  int i = min(ib, ic);
  if (ib == i && ic == i) {
    a[i] = b[pb1] + c[pc1];
  }
  else if (ib == i) {
    a[i] = b[pb1];
  }
  else {
    a[i] = c[pc1];
  }
  if (ib == i) pb1++;
  if (ic == i) pc1++;
}

while (pb1 < b1_pos[1]) {
  int i = b1_crd[pb1];
  a[i] = b[pb1++];
}

while (pc1 < c1_pos[1]) {
  int i = c1_crd[pc1];
  a[i] = c[pc1++];
}
```

AVOID WASTED WORK AND ITERATIONS

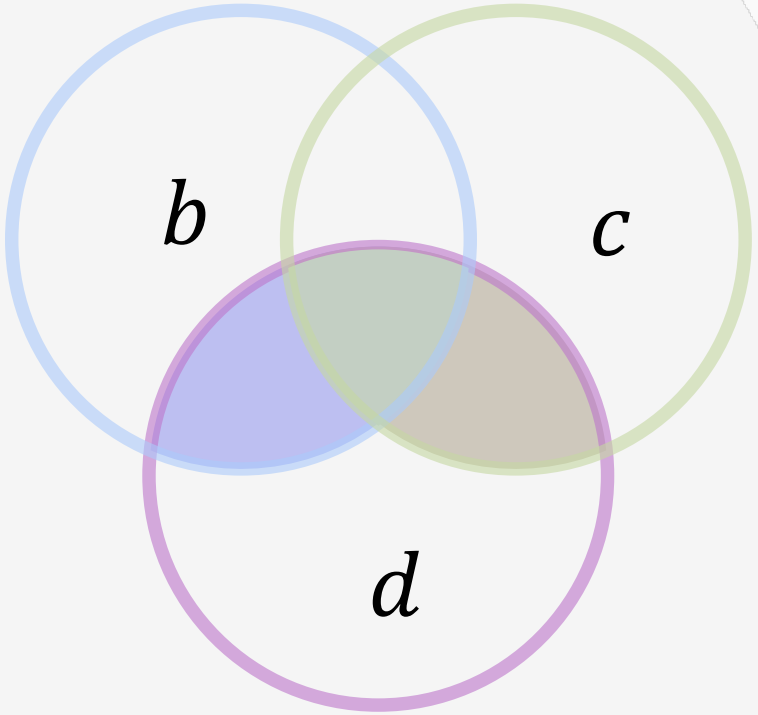


$$a_i = b_i + c_i + d_i$$

```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
int id = 0;
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
  int ib = b1_crd[pb1];
  int ic = c1_crd[pc1];
  int pd1 = id;
  int pa1 = id;
  if (ib == id && ic == id) {
    a[pa1] = b[pb1] + c[pc1] + d[pd1];
  }
  else if (ib == id) {
    a[pa1] = b[pb1] + d[pd1];
  }
  else if (ic == id) {
    a[pa1] = c[pc1] + d[pd1];
  }
  else {
    a[pa1] = d[pd1];
  }
  if (ib == id) pb1++;
  if (ic == id) pc1++;
  id++;
}

while (pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
  int ic = c1_crd[pc1];
  int id = d1_crd[pd1];
  int i = min(ic, id);
  if (ic == i && id == i) {
    a[i] = c[pc1] + d[pd1];
  }
  else if (ic == i) {
    a[i] = c[pc1];
  }
  else {
    a[i] = d[pd1];
  }
  if (ic == i) pc1++;
  if (id == i) pd1++;
}
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
  int ib = b1_crd[pb1];
  int ic = c1_crd[pc1];
  int i = min(ib, ic);
  if (ib == i && ic == i) {
    a[i] = b[pb1] + c[pc1];
  }
  else if (ib == i) {
    a[i] = b[pb1];
  }
  else {
    a[i] = c[pc1];
  }
  if (ib == i) pb1++;
  if (ic == i) pc1++;
}
```

```
while (pc1 < c1_pos[1]) {
  int ic = c1_crd[pc1];
  int pd1 = id;
  int pa1 = id;
  if (ic == id) {
    a[pa1] = c[pc1] + d[pd1];
  }
  else {
    a[pa1] = d[pd1];
  }
  if (ic == id) pc1++;
  id++;
}

while (pb1 < b1_pos[1]) {
  int ib = b1_crd[pb1];
  int pd1 = id;
  int pa1 = id;
  if (ib == id) {
    a[pa1] = b[pb1] + d[pd1];
  }
  else {
    a[pa1] = d[pd1];
  }
  if (ib == id) pb1++;
  id++;
}

while (pd1 < d1_pos[1]) {
  int id = d1_crd[pd1];
  a[id] = d[pd1];
  pd1++;
}

while (pc1 < c1_pos[1]) {
  int ic = c1_crd[pc1];
  a[ic] = c[pc1];
  pc1++;
}
while (pb1 < b1_pos[1]) {
  int ib = b1_crd[pb1];
  a[ib] = b[pb1];
  pb1++;
}

while (id < d1_dimension) {
  int pd1 = id;
  int pa1 = id;
  a[pa1] = d[pd1];
  id++;
}
```

AVOID WASTED WORK AND ITERATIONS



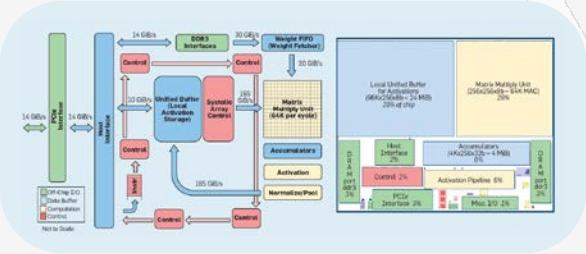
$$a_i = (b_i + c_i)d_i$$

```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
int pd1 = d1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
  int ib = b1_crd[pb1];
  int ic = c1_crd[pc1];
  int id = d1_crd[pd1];
  int i = min(ib, ic, id);
  if (ib == i && ic == i && id == i) {
    a[i] = (b[pb1] + c[pc1]) * d[pd1];
  }
  else if (ib == i && id == i) {
    a[i] = b[pb1] * d[pd1];
  }
  else if (ic == i && id == i) {
    a[i] = c[pc1] * d[pd1];
  }
  if (ib == i) pb1++;
  if (ic == i) pc1++;
  if (id == i) pd1++;
}
while (pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
  int ic = c1_crd[pc1];
  int id = d1_crd[pd1];
  int i = min(ic, id);
  if (ic == i && id == i) {
    a[i] = c[pc1] * d[pd1];
  }
  if (ic == i) pc1++;
  if (id == i) pd1++;
}

while (pb1 < b1_pos[1] && pd1 < d1_pos[1]) {
  int ib = b1_crd[pb1];
  int id = d1_crd[pd1];
  int i = min(ib, id);
  if (ib == i && id == i) {
    a[i] = b[pb1] * d[pd1];
  }
  if (ib == i) pb1++;
  if (id == i) pd1++;
}
```

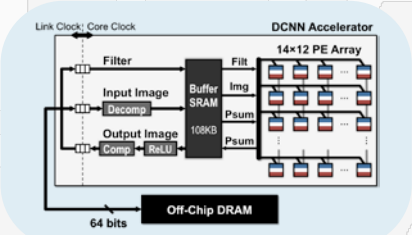
DSAS FOR ALL SPARSE TENSOR ALGEBRA

- Fixed-function Domain-Specific Architectures

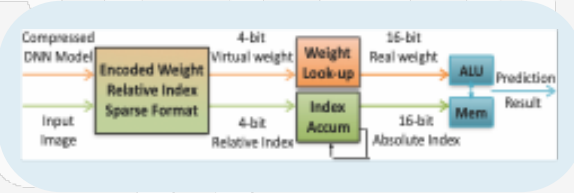
Google TPU



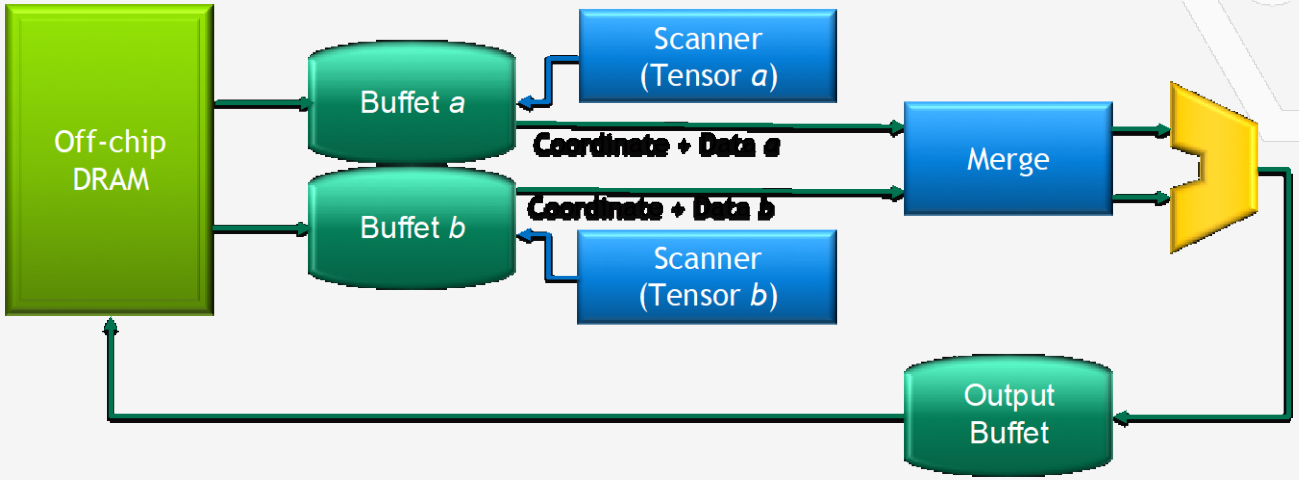
MIT Eyeriss



Stanford EIE



- Domain-Specific Architecture for all Sparse Tensor Algebra

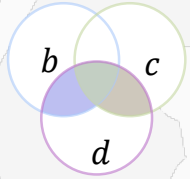


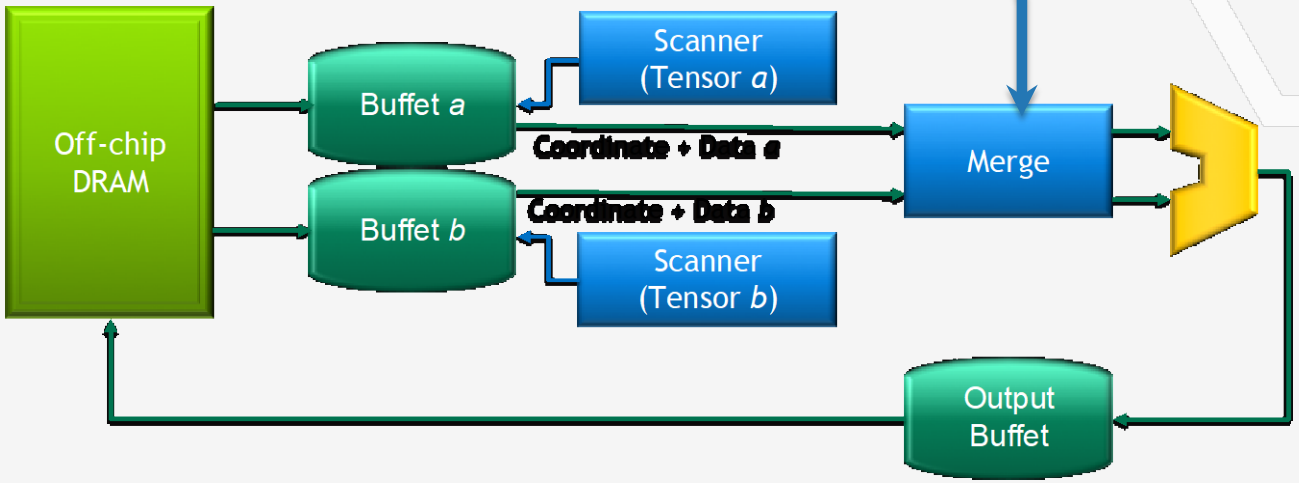
Nvidia Symphony Architecture (DARPA SDH)

Images from anandtech and Stanford

DSAS FOR ALL SPARSE TENSOR ALGEBRA

Avoid
wasted work
& iterations



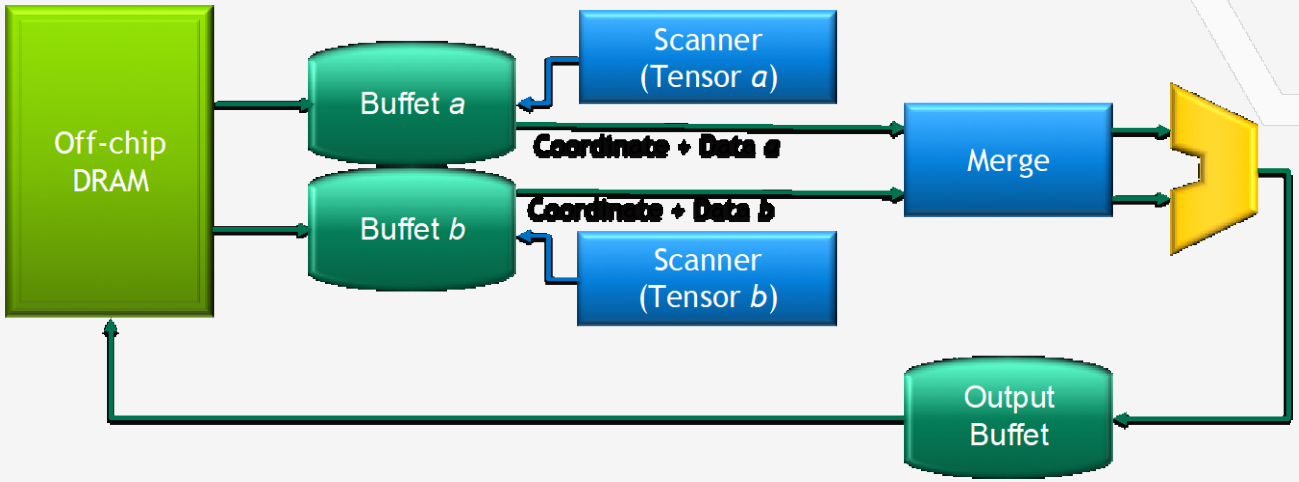


DSAS FOR ALL SPARSE TENSOR ALGEBRA

- TACO generates code for the Symphony Architecture
- Buffets, Scanners & Merge units provide acceleration
- Imperative C code for more complex outer loops

$$y = Ax + z$$

With TACO, Symphony can accelerate all tensor algebra expressions in any data format



union case #1

intersection in hardware

intersection in hardware

intersection in hardware

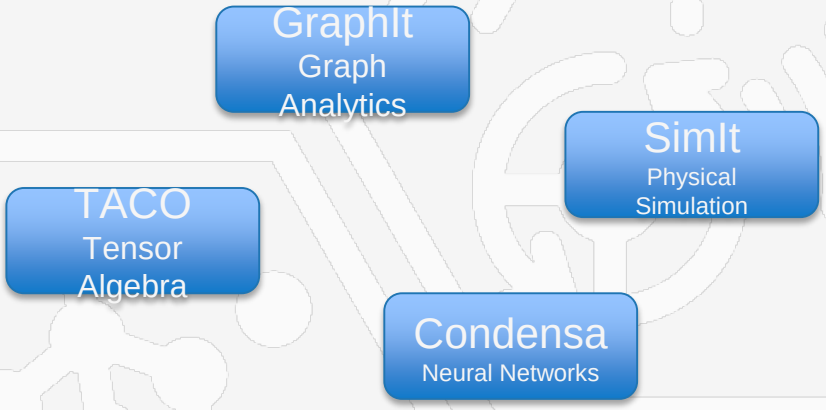
union case #2

intersection in hardware

union case #3

CHALLENGES TO A HARMONIOUS FUTURE WITH DSLS

- Proliferation of DSLs
  - Each DSL is Costly to Build and Maintain
    - Common tools for building DSL
      - Delite from Stanford
    - Common backend from DSLs to DSAs
      - MIT Tiramisu compiler middle-end, similar to LLVM compiler backend for unicores
  - Interoperability between DSLs
    - Many programs will require multiple DSLs to implement
      - Ex: Multi-physics
    - A Unified Programming Interface: Abstraction without Friction
      - Python glue with common C data structures (NumPy)
- Quality of Life of Programmers
  - Debuggers, package managers, IDE support, Foreign Function Interfaces
  - Documentation, libraries, examples and a community



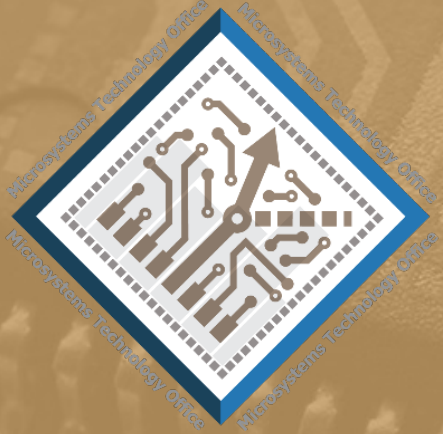
A LOT MORE PERFORMANCE LEFT TO MINE

- Current software practices are not performance efficient
  - Moore's Law era mindset of excess resources
- Case of Matrix Multiply

	Implementation	Time (s)	GFLOPS	Speedup
1	Python	25,552.48	0.005	1

CONCLUSION

- Domain Specific Architectures are the future of performance
- Domain Specific Languages are essential to DSAs
  - Make DSAs generally available for the entire domain
  - Provide high performance
- Symphony Project is building multiple DSLs
  - TACO: Sparse Tensor Algebra
  - GraphIt: Graph Analytics
  - Condensa: Neural Networks
- Looking for other DSL opportunities



ERI ELECTRONICS RESURGENCE INITIATIVE

SUMMIT

2019 | Detroit, MI | **July 15 - 17**