Safe & Productive High-Performance Programming with User-Schedulable Languages

Jonathan Ragan-Kelley / MIT CSAIL
Safe & Productive High-Performance Programming with User-Schedulable Languages

Jonathan Ragan-Kelley / MIT CSAIL
High-performance programming requires low-level control.

Therefore, it must be unsafe & unproductive.
This talk

**Halide:** user scheduling for safe, productive high performance

User-schedulable languages 2.0
Performance

What is it?
Where does it come from?
Why do we care?
Visual computing and spatiotemporal data are everywhere.
Visual computing demands orders of magnitude more computation.
Rendering: insatiable demand for computation

- Modern game:
  - 2 Mpixels
  - 10 Mpolys
  - 15 ms/frame
  - 15 ms/frame

- Tintin, Avatar:
  - 8 Mpixels
  - 5 Gpolys
  - 5 hrs/frame

6 orders of magnitude more computation

Images by Valve, Weta
The biggest data is visual

YouTube: 400 hrs uploaded / min
[Brewer 2016]
1.5 Terapixels/sec

250 M surveillance cameras,
2.5 B cell phone cameras, ...
Visual data analysis is expensive

One object detection neural net:

250 Watt GPU $\rightarrow$ 0.25 megapixels at video rate

(yolo-v3 on Tesla V100)
Programmer productivity has exploded

1990s

ORACLE

2010s

React

Python

Firebase

Amazon web services
Building high-performance systems is harder than ever

Reference:
200 lines C++

Adobe: 1500 lines
3 months of work
10x faster
My group’s research:
Compilers, systems, architectures, and algorithms for high-performance graphics & visual computing.

- Reorganize computations & data
- Capture & control dependencies
- Define & exploit domain-specific structure

- applications
- algorithms
- data structures
- languages
- compilers
- hardware
How can we increase performance and efficiency?

**Parallelism**

“Moore’s law” scaling requires exponentially more parallelism.

**Locality**

Data should move as little as possible.
Communication dominates computation in both energy and time

<table>
<thead>
<tr>
<th>Operation</th>
<th>Energy/Op (28 nm)</th>
<th>Cost (vs. ALU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU op</td>
<td>1 pJ</td>
<td>-</td>
</tr>
<tr>
<td>Load from SRAM</td>
<td>5 pJ</td>
<td>5x</td>
</tr>
<tr>
<td>Move 10mm on-chip</td>
<td>32 pJ</td>
<td>32x</td>
</tr>
<tr>
<td>Send off-chip</td>
<td>500 pJ</td>
<td>500x</td>
</tr>
<tr>
<td>Send to DRAM</td>
<td>1 nJ</td>
<td>1,000x</td>
</tr>
<tr>
<td>Send over LTE</td>
<td>&gt; 50 µJ</td>
<td>50,000,000x</td>
</tr>
</tbody>
</table>

Data from John Brunhaver, Bill Dally, Mark Horowitz
Message #1: Performance requires complex tradeoffs
Where does performance come from?

Program

Hardware

tradeoff

amount of work

communication (locality)

serial dependence (parallelism)
Message #2: organization of computation is a first-class issue
Reorganizing computation is painful

Reference:
300 lines C++

Adobe: 1500 lines

3 months of work

10x faster (vs. reference)

Same algorithm,
Different organization
Global reorganization breaks modularity

The algorithm uses 8 pyramid levels
Halide
a language and compiler with decoupled organization
Algorithm vs. Organization: 3x3 blur

\[
\text{blurH}(x, y) = \frac{\text{input}(x-1, y) + \text{input}(x, y) + \text{input}(x+1, y)}{3};
\]

\[
\text{blurV}(x, y) = \frac{\text{blurH}(x, y-1) + \text{blurH}(x, y) + \text{blurH}(x, y+1)}{3};
\]
Algorithm vs. Organization: 3x3 blur

\[
\text{blurH}(x, y) = \frac{(\text{input}(x-1, y) + \text{input}(x, y) + \text{input}(x+1, y))}{3};
\]

\[
\text{blurV}(x, y) = \frac{(\text{blurH}(x, y-1) + \text{blurH}(x, y) + \text{blurH}(x, y+1))}{3};
\]

Same algorithm, different organization
One of them is 15x faster
Algorithm vs. Organization: 3x3 blur

```
for (int y = 0; y < input.height(); y++)
  for (int x = 0; x < input.width(); x++)
    blurH(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;

for (int y = 0; y < input.height(); y++)
  for (int x = 0; x < input.width(); x++)
    blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(x, y+1))/3;
```
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);
                }
            }
        }
    }
}

8 → 0.25 ms/megapixel

30x faster (dual core x86)

Tiled, fused
Vectorized
Multithreaded
Redundant computation
Near roof-line optimum
Traditional languages conflate algorithm & organization

void box_filter_3x3(const Image &in, Image &blurV) {
    __m128 one_third = _mm_set1_epi16(1/3);
    __m128i one_third = _mm_set1_epi16(1/3);
    __m128i blurH[(256/8)*32]; // allocate tile blurH array
    Image blurH(in.width(), in.height()); // allocate blurH array

    for (int yTile = 0; yTile < in.height(); yTile += 32) {
        __m128i *blurHPtr = &blurH[yTile];
        for (int xTile = 0; xTile < in.width(); xTile += 256) {
            a = _mm_loadu_si128((__m128i *)(inPtr-1));
            b = _mm_loadu_si128((__m128i *)(inPtr));
            c = _mm_loadu_si128((__m128i *)(inPtr+1));
            sum = _mm_add_epi16(a, b, c);
            avg = _mm_add_epi16(sum, one_third);
            _mm_store_si128(blurHPtr++, avg);
            inPtr += 8;
        }
        blurHPtr = blurH;
        for (int y = 0; y < 32; y++) {
            __m128i *outPtr = (__m128i *)&blurV[yTile+y][xTile];
            for (int x = 0; x < 256; x += 8) {
                a = _mm_loadu_si128(blurHPtr);
                *outPtr = _mm_add_epi16(*outPtr, a);
                outPtr += 8;
            }
        }
    }
}

void box_filter_3x3(const Image &in, Image &blurV) {
    Image blurH(in.width(), in.height()); // allocate blurH array

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

unreadable
architecture-specific
hard to change organization or algorithm
void box_filter_3x3(const Image &in, Image &blurV) {
    __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 blurH[(256/8)*(32+2)]; // allocate tile blurH array
        for (int xTile = 0; xTile < in.width(); xTile += 256) {
            __m128i *blurHPtr = blurH;
            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(blurHPtr++, avg);
                    inPtr += 8;
                }
            }
            blurHPtr = blurH;
            for (int y = 0; y < 32; y++) {
                __m128i *outPtr = (__m128i *)&blurV[yTile+y][xTile];
                for (int x = 0; x < 256; x += 8) {
                    a = _mm_load_si128(blurHPtr-256/8);
                    b = _mm_load_si128(blurHPtr);
                    c = _mm_load_si128(blurHPtr++);
                    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                    avg = _mm_mulhi_epi16(sum, one_third);
                    _mm_store_si128(outPtr++, avg);
                }
            }
        }
    }
}

Optimized 3x3 blur in C++

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

**Optimized 3x3 blur in C++**

- **Parallelism**: distribute across threads
- **Locality**: reorganize computation: fuse two blurs, compute in tiles
- **SIMD**: parallel vectors
(Re)organizing computation is hard

Optimizing parallelism, locality requires transforming program & data structure.

What transformations are *legal*?

What transformations are *beneficial*?

*libraries don’t solve this:*

cuDNN, BLAS, MKL, OpenCV…

optimized kernels compose into inefficient pipelines (no fusion)
Halide’s answer: *decouple* algorithm from schedule

**Algorithm:** *what* is computed  
**Schedule:** *where* and *when* it’s computed
The algorithm defines pipelines as pure functions

Pipeline stages are functions from coordinates to values

Execution order and storage are unspecified
no explicit loops or arrays

3x3 blur as a Halide algorithm:

\[
\begin{align*}
\text{blurH}(x, y) &= \frac{\text{input}(x-1, y) + \text{input}(x, y) + \text{input}(x+1, y)}{3}; \\
\text{blurV}(x, y) &= \frac{\text{blurH}(x, y-1) + \text{blurH}(x, y) + \text{blurH}(x, y+1)}{3};
\end{align*}
\]
Domain scope of the programming model

All computation is over regular grids ("tensors")

Only feed-forward pipelines

Iterative computations are a (partial) escape hatch

Iteration must have bounded depth

Dependence must be inferable

User-defined clamping can impose tight bounds, when needed

Long, heterogeneous pipelines

Complex graphs, deeper than traditional stencil computations
How can we organize this computation?
Organizing a data-parallel pipeline

- input
- blurH
- blurV
Simple loops execute **breadth-first** across stages
Simple loops execute \textbf{breadth-first} across stages.

Input \rightarrow \text{blurH} \rightarrow \text{blurV}

\begin{itemize}
\item Parallelism
\end{itemize}
Breadth-first execution sacrifices locality

input

blurH

blurV

locality

parallelism

write to memory

read from memory
Breadth-first execution sacrifices locality

locality is a function of reuse distance

parallelism
Interleaved execution *(fusion)* improves locality

**fusion** *globally* interleaves computation

**input** → **blurH** → **blurV**

... reduce reuse distance from **producer** to **consumer**

**locality**

**parallelism**
Understanding dependencies

input

blurH

blurV

...
Stencils have overlapping dependencies

input

blurH

blurV
Sliding window execution sacrifices parallelism

- input
- blurH
- blurV

short reuse distance

locality

fixed order, constrains parallelism
Breaking dependencies with tiling

input

blurH

blurV

locality

parallelism
Decoupled tiles optimize **parallelism & locality**

- **input**
  - `blurH`
  - `blurV`

- **short reuse distance**
- **independence**

**locality**

**parallelism**
Breaking dependencies introduces redundant work.
Breaking dependencies introduces redundant work.
Message #1: performance requires **tradeoffs**

- **input**
  - **blurH**
  - **blurV**

---

- **trade off with**
  - **granularity of fusion**

- **redundant work**

- **locality**

- **parallelism**
  - trade off by **constraining order**
Message #2: algorithm vs. organization

order and interleaving radically alter performance of the same algorithm
dependencies limit choices of organization
A language of schedules
The schedule defines intra-stage order, inter-stage interleaving.

For each stage:

1) In what order should we compute its values?
2) When should we compute its inputs?

This is a language for scheduling choices.
The schedule defines intra-stage order, inter-stage interleaving.
Tradeoff space modeled by granularity of interleaving

coarse interleaving low locality

compute granularity

fine interleaving high locality

storage granularity

redundant computation no redundant computation
Tradeoff space modeled by granularity of interleaving

- coarse interleaving: low locality, redundant computation
- fine interleaving: high locality, no redundant computation

Compute granularity: breadth-first execution

Storage granularity:
- blurH.compute_at(root)
- .store_at(root)
Tradeoff space modeled by granularity of interleaving

- coarse interleaving: low locality
- fine interleaving: high locality

compute granularity: fine interleaving → high locality → no redundant computation
storage granularity: coarse interleaving → low locality → redundant computation

Total fusion: redundant work

```
blurH.compute_at(blurV, x).store_at(blurV, x)
```
Tradeoff space modeled by granularity of interleaving

- **Coarse interleaving** (low locality)
  - Redundant computation
  - No redundant computation
  - Capturing reuse constrains order (less parallelism)
  - Sliding window fusion

- **Fine interleaving** (high locality)
  - Compute granularity
  - Storage granularity
  - BlurH.compute_at(blurV, x)
  - .store_at(root)
**Tradeoff space modeled by granularity of interleaving**

- **Compute granularity**
  - Coarse interleaving: low locality
  - Fine interleaving: high locality

- **Storage granularity**
  - Redundant computation: redundant work
  - No redundant computation

- **Tile-level fusion**
  - BlurV.tile(xo, yo, xi, yi, W, H)
  - BlurH.compute_at(blurV, xo) .store_at(blurV, xo)
Tradeoff space modeled by granularity of interleaving

- **Coarse interleaving** (low locality): Low parallelism, high storage granularity, redundant computation.
- **Fine interleaving** (high locality): High parallelism, low storage granularity, no redundant computation.

- **Compute granularity**:
  - **Parallel sliding windows**: Coarse-grained parallelism across windows.
  - **Enlarged sliding window**: Fine-grained data-parallelism within window.
  - **Parallel enlarged sliding windows**.
Schedule primitives **compose** to create many organizations

```
blurH.compute_at_root()
blurH.compute_at(blurV, tx)
blurH.compute_at(blurV, tx).
vectorize(x, 4)
blurV.tile(x, ty, txi, yi, 8, 8)
.parallel(y)
.vectorize(xi, 4)
blurV.split(x, tx, txi, 8)
.parallel(x)
.blurV.split(y, ty, yi, 8)
.parallel(y)
.vectorize(x, 4)
.blurH.compute_at(blurV, ty)
.blurH.compute_at(blurV, ty).
store_at(blurV, yi)
.vectorize(x, 4)
```

**Schedule primitives compose to create many organizations**

- **redundant work**
- **locality**
- **parallelism**
void box_filter_3x3(const Image &in, Image &blurV) {
    __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 blurH[(256/8)*(32+2)]; // allocate tile blurH array
        for (int xTile = 0; xTile < in.width(); xTile += 256) {
            __m128i *blurHPtr = blurH;
            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(blurHPtr++, avg);
                    inPtr += 8;
                }
                blurHPtr = blurH;
            }
            for (int y = 0; y < 32; y++) {
                __m128i *outPtr = ((__m128i *)&(blurV[yTile+y][xTile]));
                for (int x = 0; x < 256; x += 8) {
                    a = _mm_load_si128(blurHPtr+((2*256)/8));
                    b = _mm_load_si128(blurHPtr+256/8);
                    c = _mm_load_si128(blurHPtr++);
                    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                    avg = _mm_mulhi_epi16(sum, one_third);
                    _mm_store_si128(outPtr++, avg);
                }
            }
        }
    }
}
The **Schedule** defines a loop nest to compute the pipeline

\[
\text{blurH}(x, y) = \frac{\text{in}(x-1, y) + \text{in}(x, y) + \text{in}(x+1, y)}{3};
\]

\[
\text{blurV}(x, y) = \frac{\text{blurH}(x, y-1) + \text{blurH}(x, y) + \text{blurH}(x, y+1)}{3};
\]

```
blurV.tile(x, y, xo, yo, xi, yi, 32, 32);
```

// for each tile
for blurV.yo:
    for blurV.xo:
        // for pixel in tile
        for blurV.yi:
            for blurV.xi:
                compute blurV
The **Schedule** defines a **loop nest** to compute the pipeline:

\[
\text{blurH}(x, y) = \frac{\text{in}(x-1, y) + \text{in}(x, y) + \text{in}(x+1, y)}{3};
\]

\[
\text{blurV}(x, y) = \frac{\text{blurH}(x, y-1) + \text{blurH}(x, y) + \text{blurH}(x, y+1)}{3};
\]

```
blurV.tile(x, y, xo, yo, xi, yi, 256, 32);
blurH.compute_at(blurV, xo);
```

// for each tile
```
for blurV.yo:
    for blurV.xo:
        // for pixel in tile
        for blurV.yi:
            for blurV.xi:
                compute blurV
```
The **Schedule** defines a **loop nest** to compute the pipeline

\[
\begin{align*}
\text{blurH}(x, y) &= \frac{\text{in}(x-1, y) + \text{in}(x, y) + \text{in}(x+1, y)}{3}; \\
\text{blurV}(x, y) &= \frac{\text{blurH}(x, y-1) + \text{blurH}(x, y) + \text{blurH}(x, y+1)}{3};
\end{align*}
\]

\[\text{blurV.tile}(x, y, xo, yo, xi, yi, 256, 32);\]
\[\text{blurH.compute_at(blurV, xo);}\]

// for each tile
for blurV.yo:
  for blurV.xo:
    // for pixel in required tile
    for blurH.y:
      for blurH.x:
        compute blurH
    // for pixel in tile
    for blurV.yi:
      for blurV.xi:
        compute blurV

The **Schedule** defines a loop nest to compute the pipeline

\[
\text{blurH}(x, y) = (\text{in}(x - 1, y) + \text{in}(x, y) + \text{in}(x + 1, y))/3;
\]

\[
\text{blurV}(x, y) = (\text{blurH}(x, y - 1) + \text{blurH}(x, y) + \text{blurH}(x, y + 1))/3;
\]

blurV.tile\( (x, y, xo, yo, xi, yi, 256, 32) \).parallel\( (yo) \);

blurH.compute\_at\( (\text{blurV}, xo) \).vectorize\( (x, 8) \);

// for each tile

```plaintext
parallel for blurV.yo:
  for blurV.xo:
    // for pixel in required tile
    for blurH.y:
      vec for blurH.x:
        compute blurH<8>
    // for pixel in tile
    for blurV.yi:
      for blurV.xi:
        compute blurV
```
Func box_filter_3x3(Func in) {
    Func blurH, blurV;
    Var x, y, xi, yi;

    // The algorithm - no storage, order
    blurH(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
    blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(x, y+1))/3;

    // The schedule - defines order, locality; implies storage
    blurV.tile(x, y, xi, yi, 256, 32)
        .vectorize(xi, 8).parallel(y);
    blurH.compute_at(blurV, x).store_at(blurV, x).vectorize(x, 8);

    return blurV;
}
# Halide

0.25 ms/megapixel

```c
void box_filter_3x3(const Image &in, Image &blurV)
{
    __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 *blurHPtr = blurH;
        for (int y = -1; y < 32; 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_loadu_si128((__m128i*)(inPtr));
                sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                avg = _mm_mulhi_epi16(sum, one_third);
                _mm_storeu_si128(blurHPtr++, avg);
                inPtr += 8;
            }
        }
        blurHPtr = blurH;
        for (int y = 0; y < 32; y++)
        {
            __m128i *outPtr = (__m128i*)(&blurV[yTile+y][xTile]);
            for (int x = 0; x < 256; x += 8)
            {
                a = _mm_loadu_si128(blurHPtr+2*(256/8));
                b = _mm_loadu_si128(blurHPtr+256/8);
                c = _mm_loadu_si128(blurHPtr++);
                sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                avg = _mm_mulhi_epi16(sum, one_third);
                _mm_storeu_si128(outPtr++, avg);
            }
        }
    }
}
```

# C++

0.25 ms/megapixel

```c++
void box_filter_3x3(const Image &in, Image &blurV) {
    _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 *blurHPtr = (256/8)*(32-2));
        // allocate tile blurH array
        for (int xTile = 0; xTile < in.width(); xTile += 256)
        {
            _m128i *blurHPtr = blurH;
            for (int y = -1; y < 32; 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_loadu_si128((__m128i*)(inPtr));
                    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                    avg = _mm_mulhi_epi16(sum, one_third);
                    _mm_storeu_si128(blurHPtr++, avg);
                    inPtr += 8;
                }
            }
            blurHPtr = blurH;
            for (int y = 0; y < 32; y++)
            {
                _m128i *outPtr = (__m128i*)(&blurV[yTile+y][xTile]));
                for (int x = 0; x < 256; x += 8)
                {
                    a = _mm_loadu_si128(blurHPtr+2*(256/8));
                    b = _mm_loadu_si128(blurHPtr+256/8);
                    c = _mm_loadu_si128(blurHPtr++);
                    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                    avg = _mm_mulhi_epi16(sum, one_third);
                    _mm_storeu_si128(outPtr++, avg);
                }
            }
        }
    }
}```
Organization requires global tradeoffs

LUT: look-up table
\[ O(x,y,k) \leftarrow \text{lut}(I(x,y) - k\sigma) \]

ADD: addition
\[ O(x,y) \leftarrow I_1(x,y) + I_2(x,y) \]

SUB: subtraction
\[ O(x,y) \leftarrow I_1(x,y) - I_2(x,y) \]

DDA: data-dependent access
\[
\begin{align*}
k &\leftarrow \text{floor}(I_1(x,y) / \sigma) \\
\alpha &\leftarrow (I_1(x,y) / \sigma) - k \\
O(x,y) &\leftarrow (1 - \alpha) I_2(x,y,k) + \alpha I_2(x,y,k+1)
\end{align*}
\]

The algorithm uses 8 pyramid levels

input

3x3 box filter

[Paris et al. 2010, Aubry et al. 2011]
Local Laplacian Filters in Adobe Photoshop

Adobe: 1500 lines
expert-tuned C++
multi-threaded, SSE
3 months of work
10x faster than original C++

Halide: 60 lines
1 day

Halide vs. Adobe: 2x faster on same CPU
Adobe: 1500 lines
expert-tuned C++
multi-threaded, SSE
3 months of work
10x faster than original C++

Halide: 60 lines
1 day

Halide vs. Adobe:
2x faster on same CPU

The algorithm uses 8 pyramid levels
Adobe: 1500 lines
expert-tuned C++
multi-threaded, SSE
3 months of work
10x faster than original C++

Halide: 60 lines
1 day

Halide vs. Adobe:
2x faster on same CPU

The algorithm uses 8 pyramid levels

10x faster on GPU
Real-world adoption
open source at http://halide-lang.org

Google
> 2000 pipelines
10s of kLOC in production

Pixel HDR+
“Best smartphone camera ever”
User-schedulable languages are spreading

HALIDE  tvm

TACO  GraphIt
User-Schedulable Languages 2.0

Programming New HW & Verifying Optimizations

[PLDI 2022 + POPL 2022]
Machine Learning Programmer
Framework/Compiler Developer
Performance Engineers
Computer Architects

Intel MKL
CoreML
BLAS
cuDNN
Apple Neural Engine
GraphCore IPU
NVIDIA GPU
Google TPU
# The ML Stack

<table>
<thead>
<tr>
<th>Applications / Models</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framework / Compiler</td>
</tr>
<tr>
<td>HPC Library</td>
</tr>
<tr>
<td>Accelerators</td>
</tr>
<tr>
<td>Applications / Models</td>
</tr>
<tr>
<td>-----------------------</td>
</tr>
<tr>
<td>HPC Library</td>
</tr>
</tbody>
</table>

95%+ of computation
The ML Stack Stovepipe

<table>
<thead>
<tr>
<th>Applications / Models</th>
<th>Framework / Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HPC Library</td>
</tr>
<tr>
<td></td>
<td>Accelerators</td>
</tr>
</tbody>
</table>

| Usual Hardware |

95%+ of computation
95%+ of computation

Applications / Models

Framework / Compiler

HPC Library

Accelerators

Usual Hardware
Consequence 1: Portability Interface

<table>
<thead>
<tr>
<th>Applications / Models</th>
<th>Framework / Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPC Library</td>
<td>GEMM, Conv2d, etc.</td>
</tr>
<tr>
<td>Accelerators</td>
<td>Usual Hardware</td>
</tr>
</tbody>
</table>
## The ML Stack Stovepipe

### Consequence 1: Portability Interface

<table>
<thead>
<tr>
<th>Applications / Models</th>
<th>Framework / Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HPC Library</strong></td>
<td><strong>Usual Hardware</strong></td>
</tr>
</tbody>
</table>

- **ISA**
- **GEMM, Conv2d, etc.**
**The ML Stack Stovepipe**

Consequence 2: HPC Libs are on “Critical Path”

<table>
<thead>
<tr>
<th>Applications / Models</th>
<th>Framework / Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GEMM, Conv2d, etc.</td>
</tr>
<tr>
<td>HPC Library</td>
<td>ISA</td>
</tr>
<tr>
<td>Accelerators</td>
<td>Usual Hardware</td>
</tr>
</tbody>
</table>
What’s Special About Writing HPC Libraries?
What’s Special About Writing HPC Libraries?

Usual Programs

Language & Compiler

HW Target

Programs
What’s Special About Writing HPC Libraries?

Usual Programs

- Programs
- Language & Compiler
- HW Target

HPC Libraries

- API Specification
- Kernel “Scheduling”
- HW Target
The Trouble With Automation

HPC Libraries

API Specification

Kernel “Scheduling”

HW Target
The Trouble With Automation

HPC Libraries

- API Specification
- Kernel “Scheduling”
- HW Target
The Trouble With Automation

HPC Libraries

API Specification

Kernel “Scheduling”

HW Target
Exocompilation

a compiler/language design that externalizes parts of the compiler in order to give programmers more control
Exocompilation

a compiler/language design that externalizes parts of the compiler in order to give programmers more control

HW Targets as Libraries | User Scheduling
def gemm(N : size, M : size, K : size,
    A : f32[N,K], B : f32[K,M], C : f32[N,M]):
    for i in seq(0,N):
        for j in seq(0,M):
            for k in seq(0,K):
                C[i,j] += A[i,k] * B[k,j]
Performance Gains from Scheduling

def gemm(N : size, M : size, K : size,
         A : f32[N,K], B : f32[K,M], C : f32[N,M]):
    for i in seq(0,N):
        for j in seq(0,M):
            for k in seq(0,K):
                C[i,j] += A[i,k] * B[k,j]

Naive Matrix Multiply

88.1% of peak throughput!

Matrix Multiply Optimized for AVX-512

0.94% of peak throughput
Exocompilation is a compiler/language design that externalizes parts of the compiler in order to give programmers more control.
HW Backends as User-Level Libraries

User Code

App 1  App 2

Traditional Compiler

HW 1  HW 2

Compiler Code
HW Backends as User-Level Libraries

User Code

App 1  App 2  ⋮

Traditional Compiler

HW 1  HW 2  ⋮

Compiler Code

App 1  ⋮  HW 1  ⋮

Exocompiler
HW Backends as User-Level Libraries

User Code

App 1  App 2 ⋯

Traditional Compiler

HW 2

⋯

Compiler Code

Exocompiler

⋯

App 1
HW Backends as User-Level Libraries

User Code

Traditional Compiler

Compiler Code

Exocompiler
HW Backends as User-Level Libraries

User Code

- Instructions
- Memories
- Configuration State

Compiler Code

App 1

Exocompiler
HW Backends as User-Level Libraries

User Code

- Instructions
- Memories
- Configuration State

Compiler Code

Exocompiler

App 1

...
Instructions are Procedures

- Instructions
- Memories
- Configuration State

```
def main(...):
    ...
    instr_A(...)
    ...
```

```
@instr("ABCD")
def instr_A(...):
    ...
```
Instructions are Procedures

- Instructions
- Memories
- Configuration State

```python
def main(...):
    ...
    instr_A(...)
    ...

@instr("ABCD")
def instr_A(...):
    ...

void main(...) {
    ...
    ABCD
    ...
}
```
Instructions are Procedures

- Instructions
- Memories
- Configuration State

```python
def main(...):
    ...
    instr_A(...)
    ...
```

```python
@instr("ABCD")
def instr_A(...):
    ...
```
Exocompilation

a compiler/language design that externalizes parts of the compiler in order to give programmers more control

HW Targets as Libraries

User Scheduling
def gemm(N : size, M : size, K : size,
         A : f32[N,K], B : f32[K,M], C : f32[N,M]):
    for i in seq(0,N):
        for j in seq(0,M):
            for k in seq(0,K):
                C[i,j] += A[i,k] * B[k,j]
User Scheduling

def gemm(N : size, M : size, K : size,
       A : f32[N,K], B : f32[K,M], C : f32[N,M]):
    for i in seq(0,N):
        for j in seq(0,M):
            for k in seq(0,K):
                C[i,j] += A[i,k] * B[k,j]
def gemm(N : size, M : size, K : size,
       A : f32[N,K], B : f32[K,M], C : f32[N,M]
):
    for i in seq(0,N):
        for j in seq(0,M):
            for k in seq(0,K):
                C[i,j] += A[i,k] * B[k,j]
User Scheduling
A new approach

Scheduling as Rewriting, not Annotation

Scheduling of Sub-procedures

def gemm(N : size, M : size, K : size,
    A : f32[N,K], B : f32[K,M], C : f32[N,M]
):
    for i in seq(0,N):
        for j in seq(0,M):
            for k in seq(0,K):
                C[i,j] += A[i,k] * B[k,j]
Scheduling as Program Rewriting

Exo Program 1  →  Exo Program 2  →  ...  →  Exo Program N
Scheduling as Program Rewriting
Scheduling as Program Rewriting
Scheduling as Program Rewriting

swap
Scheduling Sub-procedures
Scheduling Sub-procedures

inline
Scheduling Sub-procedures

replace

+
Scheduling Sub-procedures

```python
@instr("ABCD")
def instr_A(...):
    ...
```
Scheduling Operations in Exo

- swap
- inline
- replace
- reorder
- split
- unroll
- set_memory
- set_precision
- bind_expr
- stage_mem
- bind_config
- reorder_dim
- expand_dim
- add_guard
- lift_alloc
- fission_after
- reorder_stmts
- config_write
- fuse_loop
- lift_if
- partition
- remove_loop
- ...

...
Scheduling Operations in Exo

Safety checked using SMT verification

- swap
- reorder
- lift_alloc
- stage_mem
- bind_config
- reorder_dim
- expand_dim
- add_guard
- partition
- remove_loop
- ...
Exo Performance Results
AVX-512 MatMul
Matches best possible performance

Problem Size $N(A, B, C \in \mathbb{R}^{N \times N})$.
GEMMINI MatMul

2.3x - 3.7x faster than original HPC Library

% of peak throughput

N x M x K

<table>
<thead>
<tr>
<th>N x M x K</th>
<th>Old Lib</th>
<th>Exo Lib</th>
</tr>
</thead>
<tbody>
<tr>
<td>512 x 512 x 512</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12544 x 256 x 64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12544 x 64 x 256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3136 x 512 x 128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3136 x 128 x 512</td>
<td></td>
<td></td>
</tr>
<tr>
<td>784 x 1024 x 256</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

% of peak throughput
GEMMINI Conv

Similar Story for Convolution Layers

W x H x IC x OC | Old Lib | Exo Lib
---|---|---
56 x 56 x 64 x 64 A | | 
28 x 28 x 128 x 128 B | | 
14 x 14 x 256 x 256 C | | 

% of peak throughput

0% 25% 50% 75% 100%
Exo vs. Existing - Lines of Code

AVX-512 MatMul
- Exo: 0
- MKL or GEMMINI (old): >1650

GEMMINI MatMul
- Exo: 0
- MKL or GEMMINI (old): 4.7x

GEMMINI Conv
- Exo: 0
- MKL or GEMMINI (old): 6.4x
HW / SW Co-Design
Benefits to Software Maintenance

ISA Change

5 lines of Exo code changed

46 lines of old handwritten GEMMINI library changed
GEMMINI MatMul + Conv
HW / SW Co-Design

MatMul

Old Lib | Exo Lib

0% | 25% | 50% | 75% | 100%

Conv

Old Lib | Exo Lib

0% | 25% | 50% | 75% | 100%
Exocompilation for Productive Programming of Hardware Accelerators

https://exo-lang.dev
for i in seq(0,N):
    for j in seq(0,M):
        for k in seq(0,K):
            C[i,j] += A[i,k] * B[k,j]
Observations From Exo

for i in seq(0,N):
    for j in seq(0,M):
        for k in seq(0,K):
            C[i,j] += A[i,k] * B[k,j]
Functional Scheduling

for i in seq(0,N): 
    for j in seq(0,M): 
        for k in seq(0,K): 
            C[i,j] += A[i,k] * B[k,j]
Functional Scheduling

for $i$ in seq(0,N):
    for $j$ in seq(0,M):
        for $k$ in seq(0,K):
            $C[i,j] += A[i,k] \cdot B[k,j]$
First Formally Verified User-Schedulable Language

Coq

\[ N \sum_{i=0}^{M} \sum_{j=0}^{K} A[i,k] \cdot B[k,j] \]

Lemmas \quad \leftrightarrow \quad Scheduling Rewrites

Interactive Proving \quad \leftrightarrow \quad Interactive Scheduling

[POPL 2022]
User-Schedulable Languages 2.0

Programming New HW & Verifying Optimizations

[PLDI 2022 + POPL 2022]
Acknowledgements

Andrew Adams  Adam Chlipala  Steven Johnson  Ravi Mullapudi
Luke Anderson  Frédo Durand  Shoaib Kamil  Sylvain Paris
Saman Amarasinghe  Kayvon Fatahalian  Marc Levoy  Alex Reinking
Connelly Barnes  Hasan Genc  Tzu-Mao Li  Dillon Sharlet
Gilbert Bernstein  Michael Gharbi  Amanda Liu  Zalman Stern
Yuka Ikarashi  Karima Ma
Thank you!
## Acknowledgements

<table>
<thead>
<tr>
<th>Andrew Adams</th>
<th>Steven Johnson</th>
</tr>
</thead>
<tbody>
<tr>
<td>Luke Anderson</td>
<td>Shoaib Kamil</td>
</tr>
<tr>
<td>Saman Amarasinghe</td>
<td>Marc Levoy</td>
</tr>
<tr>
<td>Connelly Barnes</td>
<td>Tzu-Mao Li</td>
</tr>
<tr>
<td>Gilbert Bernstein</td>
<td>Amanda Liu</td>
</tr>
<tr>
<td>Adam Chlipala</td>
<td>Karima Ma</td>
</tr>
<tr>
<td>Frédo Durand</td>
<td>Ravi Mullapudi</td>
</tr>
<tr>
<td>Kayvon Fatahalian</td>
<td>Sylvain Paris</td>
</tr>
<tr>
<td>Hasan Genc</td>
<td>Alex Reinking</td>
</tr>
<tr>
<td>Michael Gharbi</td>
<td>Dillon Sharlet</td>
</tr>
<tr>
<td>Yuka Ikarashi</td>
<td>Zalman Stern</td>
</tr>
</tbody>
</table>
My group’s research:

Capture & control **dependencies**
Reorganize computations & data

Define & exploit structure

Image Processing
3D Graphics
Machine Learning

| applications |
| algorithms |
| data structures |
| languages |
| compilers |
| hardware |