High performance data processing with Halide
Roel Jordans
High performance computing
Architecture Trends

40 Years of Microprocessor Trend Data

Transistors (thousands)

Single-Thread Performance (SpecINT x 10^3)

Frequency (MHz)

Typical Power (Watts)

Number of Logical Cores

Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten

New plot and data collected for 2010-2015 by K. Rupp
Processing platform architectures

- GPU
  - Cache
    - Local memory
  - CPU
    - Cache
- CPU
  - Cache
  - Memory

Increasing latency / energy
Programming challenges

- Data movement is costly
  - Requires early planning of data distribution
  - May require explicit moves from CPU to GPU memory
- Traditional programming languages (like C) are not designed for this kind of architecture
  - Require library support
  - Highly customized code
  - Vendor specific, low portability
An example: 3x3 box filter

• A simple, two-stage imaging pipeline: \textbf{3x3 blur}.

Basic function: a summation over a 3x3 area:

\[
\begin{align*}
    bx(x, y) &= \text{in}(x - 1, y) + \text{in}(x, y) + \text{in}(x + 1, y) \\
    by(x, y) &= bx(x, y - 1) + bx(x, y) + bx(x, y + 1)
\end{align*}
\]

• We leave out the averaging step.
**Blur: Inlined implementation**

**C code**

```c
int in[W*H];
int by[W*H];
for(int y=1; y<(H-1); y++){
    for(int x=1; x<(W-1); x++){
        by[x + y*W] = in[(x-1) + (y-1)*W] + in[(x-1) + y*W] + in[(x-1) + (y+1)*W] +
                     in[x + (y-1)*W] + in[x + y*W] + in[x + (y+1)*W] +
                     in[(x+1) + (y-1)*W] + in[(x+1) + y*W] + in[(x+1) + (y+1)*W];
    }
}
```

- 9 loads per output pixel
- 8 additions per output pixel
- Minimal memory footprint
- Completely parallelizable (independent pixels)
- Unnecessary recomputation
Blur: Stored implementation

*C code*

```c
int in[W*H];
int bx[W*H];
int by[W*H];
for(int y=0; y<H; y++){
    for(int x=1; x<(W-1); x++){
        bx[x + y*W] = in[x-1 + y*W] + in[x + y*W] + in[x+1 +y*W];
    }
}
for(int y=1; y<(H-1); y++){
    for(int x=1; x<(W-1); x++){
        by[x + y*W] = bx[x + (y-1)*W] + bx[x + y*W] + bx[x+ (y+1)*W];
    }
}
```

- 6 loads, 1 store per output pixel
- 4 additions per output pixel
- Very low locality (big buffer)
- No recomputation
- Still parallelizable
**Blur: Fused pipeline**

**C code**

```c
int in[W*H];
int bx[W*H];
int by[W*H];
for(int y=0; y<2; y++){
    for(int x=1; x<(W-1); x++){
        bx[x + y*W] = in[x-1 + y*W] + in[x + y*W] + in[x+1 + y*W];
    }
}
for(int y=1; y<(H-1); y++){
    for(int x=1; x<(W-1); x++){
        bx[x + (y+1)*W] = in[x-1 + (y+1)*W] + in[x + (y+1)*W] + in[x+1 + (y+1)*W];
        by[x + y*W] = bx[x + (y-1)*W] + bx[x + y*W] + bx[x + (y+1)*W];
    }
}
```

- 6 loads, 1 store per output pixel
- 4 additions per output pixel
- Not directly parallelizable
- High locality (producer, consumer moved together)
- No recomputation
C code

int in[W*H];
int bx[W*3];
int by[W*H];
for(int y=0; y<2; y++){
    for(int x=1; x<(W-1); x++){
        bx[x + y*W] = in[x-1 + y*W] + in[x + y*W] + in[x+1 + y*W];
    }
}
for(int y=1; y<(H-1); y++){
    for(int x=1; x<(W-1); x++){
        bx[(x + (y+1)*W)%3] = in[x-1 + (y+1)*W] + in[x + (y+1)*W] + in[x+1 + (y+1)*W];
        by[x + y*W] = bx[(x + (y-1)*W)%3] + bx[(x + y*W)%3] + bx[(x + (y+1)*W)%3];
    }
}

- Same results as last slide, but:
- With a smaller intermediate buffer (W*3 instead of W*H)
Data mapping freedom

• Two extremes
  • Inline everything → Lots of computations
  • Store everything → Lots of memory required

• Many options in between
  • Can be tuned to match memory hierarchy!
  • Can result in really complex loop structures
Next level optimizations

C allows us to specifically program the execution order

Many optimizations:

• Loop fusion, storage folding, tiling, multi-threading, vectorization, ...
• Most **obscure functionality**
• Most are **architecture specific**
• Requires **rewriting and debugging** to optimize
• Exploration of optimizations is **increasingly difficult**
Blur optimized

```c
#pragma omp parallel for
for (int yTile = 0; yTile < out.height(); yTile += 32) {
    __m128i a, b, c, sum, avg;
    m128i tmp[(128/8) * (32 + 2)];
    Func blur_3x3(Func input) {
        Func blur_x, blur_y;
        Var x, y, xi, yi;

        // The algorithm - no storage or order
        bx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y));
        by(x, y) = (bx(x, y-1) + bx(x, y) + bx(x, y+1));

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

        return by;
    }

    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
    avg = _mm_mulhi_epi16(sum, one_third);
    _mm_store_si128(outPtr++, avg);
}
```

**Halide (complete implementation)**

**C (partial implementation)**
Halide?

• A domain specific language (DSL) targeting image processing pipelines
  • Embedded in C++ → Uses an pre-existing compiler for most of the heavy lifting
  • Available for many target architectures (x86, ARM, CUDA, ...)
  • Support from industry: Google, Adobe
Halide!

- **Main idea**
  - Decouple algorithm definition from optimization schedule
    → Apply optimizations without complicating the code

- **Result**
  - Easier and faster design space exploration
  - Improved readability and portability
    → For a new architecture we should only change the schedule
**Blur: Halide**

Horizontal blur -> Vertical blur

\[
\begin{align*}
bx(x, y) &= \text{in}(x - 1, y) + \text{in}(x, y) + \text{in}(x + 1, y) \\
by(x, y) &= bx(x, y - 1) + bx(x, y) + bx(x, y + 1)
\end{align*}
\]

Func \(\text{in}, \ bx, \ by;\)

Var \(x, \ y;\)

\[
\begin{align*}
\text{bx}(x, y) &= \text{in}(x-1, y) + \text{in}(x, y), + \text{in}(x+1, y); \\
\text{by}(x, y) &= \text{bx}(x, y-1) + \text{bx}(x, y) + \text{bx}(x, y+1); \\
\text{by}.\text{realize}(10,10); \quad // \text{build and execute the loop nest over a 10x10 area}
\end{align*}
\]
Blur: Halide

Func bx, by, in;
Var x, y;
bx(x, y) = in(x-1, y) + in(x, y) + in(x+1, y);
by(x, y) = bx(x, y-1) + bx(x, y) + bx(x, y+1);

Note that in the body, there is no notion of:
• time (execution order).
• space (buffer assignment, image size, memory allocation)
• hardware (because no time and space)

• a very clear, concise and readable algorithm.
• we have not chosen any optimization strategy yet.
  • eg. we can use this same starting point on any target architecture.
  • (in C, a naïve implementation would already require scheduling decisions)
Scheduling

**Halide**

```
Func bx, by, in;
Var x, y;

bx(x, y) = in(x-1, y) + in(x, y) + in(x+1, y);
by(x, y) = bx(x, y-1) + bx(x, y) + bx(x, y+1);
```

```
by.realize(10,10);
```

- Internally, Halide converts this functional representation to a C-like loop nest.
- By default, if nothing else is done, everything is **inlined**.
Scheduling

**Halide**

```
Func bx, by, in;
Var x, y;
bx(x, y) = in(x-1, y) + in(x, y) + in(x+1, y);
by(x, y) = bx(x, y-1) + bx(x, y) + bx(x, y+1);

bx.compute_root();
by.realize(10,10);
```

`compute_root()`: compute and store all outputs of a (producer) function before starting computation of the next.
**Halide**

```
Func bx, by, in;
Var x, y;

bx(x, y) = in(x-1, y) + in(x, y) + in(x+1, y);
by(x, y) = bx(x, y-1) + bx(x, y) + bx(x, y+1);

bx.compute_at(by, y);
by.realize(10,10);
```

- `compute_root()` is actually a special case of `compute_at()`.

- `compute_at(by, y)` means: “*Whenever stage by starts an iteration of the y loop, first calculate the pixels of stage bx that will be consumed.*”

- In other words: computation of `bx` is fused at the loop over `y` of `by`.

Not completely equivalent to our initial fused version
Scheduling

**Halide**

```plaintext
Func bx, by, in;
Var x, y;

bx(x, y) = in(x-1, y) + in(x, y) + in(x+1, y);
by(x, y) = bx(x, y-1) + bx(x, y) + bx(x, y+1);

bx.compute_at(by, y);
bx.store_root();

by.realize(10, 10);
```

- For this, we can separate computation from storage using `store_at()` and `store_root()`.
- `bx.store_root()` means: "Allocate the buffer for bx outside the loop nest."

Halide automatically applies storage folding as well!
Many more scheduling options

• We looked at the syntax which **interleaves** computation between stages.

• There is also syntax which changes the **order of computation within a single stage**:
  - **Reorder** loop variables → `by.reorder(y,x);`
  - **Split** loop variables into inner and outer → `by.split(x, xout, xin, 4);`
  - **Tiling** is a just combination of the above:
    - `by.split(x, xout, xin, 4);`
    - `by.split(y, yout, yin, 4);`
    - `by.reorder(xin, yin, xout, yout);`
    - *Because this is so common, **syntactic sugar** (a “shortcut”) is offered:*
      - `by.tile(x, y, xout, xin, yout, yin, 4, 4);`

• **Execute loop iterations in parallel** using multi-threading:
  - `by.parallel(x);` //executes each x iteration simultaneously in threads

• Turn a loop into a **(series of) vector operation(s):**
  - `by.vectorize(xin);` //loop over xin, which has 4 iterations, is vectorized
  - `by.vectorize(x, 4);` //shortcut: split x into out and in of 4, then vectorize in
Many scheduling options

Func gradient;
Var x, y, xout, xin, yout, yin;

//1-line algorithm definition:
gradient(x,y) = x+y;

//this is equivalent to
//gradient.tile(x, y, xout, xin, yout, yin, 2,
gradient.split(x, xout, xin, 2);
gradient.split(y, yout, yin, 2);
gradient.reorder(xin, yin, xout, yout);

gradient.vectorize(xin);

gradient.parallel(yout).parallel(xout);
Larger program: Local Laplacian filters

- 99 different stages
- many different stencils
- large data-dependent resampling.

Reference: 300 lines in C++
Adobe (C++): 1500 lines expert-optimized, multi-threaded, SIMD, 10x faster 3 months of work
Intern (Halide): 60 lines, 2x faster (vs. expert), 1 day GPU version, 9x faster (vs. expert)
Auto-scheduling

Halide now includes an **auto-scheduler**

- User provides an estimate of the **problem size**
  - e.g. by.estimate(x,0,1920).estimate(y,0,1024);
- Compiler attempts to **automatically** generate an optimization schedule for the pipeline
  - Tiling
  - Fusion
  - Vectorization
  - Parallelization
- User can inspect the schedule and optimize it further
Limitations

• As mentioned Halide is **domain-specific** to image processing. It can be less suitable for other workloads because:
  
  • Not **Turing-complete** (no full recursion)
  • Only iterates over **rectangular domains**
  • Scheduling model only covers **typical image processing optimizations**

• But this is the point of domain-specific languages:
  
  • If we aim to cover everything, we will get something flexible like C again!
Observations

• With Halide, the algorithm definition is more clear and concise than with C.
  • Being separated from the optimization strategy
• Transformations that would normally take a lot of effort are done in just a few separate scheduling statements
  • Saves time
  • Guaranteed correctness
  • Automatic handling of edge conditions and storage folding
• With Halide, we can easily port the algorithm to a different architecture
  • As long as a Halide back-end exists for that architecture
  • Code is hardware-independent
  • For good performance on the new architecture → re-write the schedule