EEP590 Spring 2022

Deep Learning for Embedded Real Time Intelligence

Lecture 8: Introduction to Halide Programing Language

Prof. Richard Shi Department of Electrical and Computer Engineering (cjshi@uw.edu)

ELECTRICAL & COMPUTER ENGINEERING

Disclaims

This slide is prepared based on the Halide resources available on the internet with contributions from many people. Here is a partial list:

Ⓟ https://halide-lang.org/
➔

Jonathan Ragan-Kelley Ph.D. dissertation, MIT, May 2014. (The design and implementation of the Halide language and compiler)

- "Optimizing Image Processing Algorithms With Halide", Wednesday 12/9/20 09:00am: Posted By Hsin-I Hsu, Qualcomm (https://developer.qualcomm.com/blog/optimizing-image-processing-algorithms-halide)

Outline

1. A Motivational Example for Halide
2. Halide Programming Languages

Often used to go from image feature map -> final output or map image features to a single vector

Eliminates spatial information

Intelligent Architectures

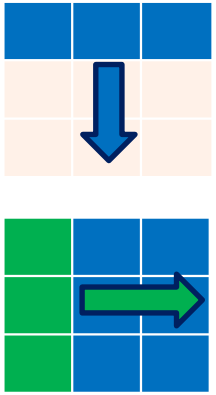
5LIL0

Introduction to Halide

Savvas Sioutas & Henk Corporaal
www.ics.ele.tue.nl/~heco/courses/IA
h.corporaal@tue.nl
TUEindhoven
2019

A Motivational Example: 3x3 Box Filter

- A simple, two-stage imaging pipeline: **3x3 blur**.



Basic function: a summation over a 3x3 area:

$$bx(x,y)=in(x-1,y)+in(x,y)+in(x+1,y)$$

$$by(x,y)=bx(x,y-1)+bx(x,y)+bx(x,y+1)$$

- We leave out the averaging step (i.e. dividing by 3).

Blur Inline

Linearized the 2-d index into a 1-d index

C code

```
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
- Completely parallelizable (independent pixels)
- Unnecessary recomputation

Blur Inline

C code

```
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
- Completely parallelizable (independent pixels)
- Unnecessary recomputation

Blur Stored Implementation

C code

```
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 Stored Implementation

C code

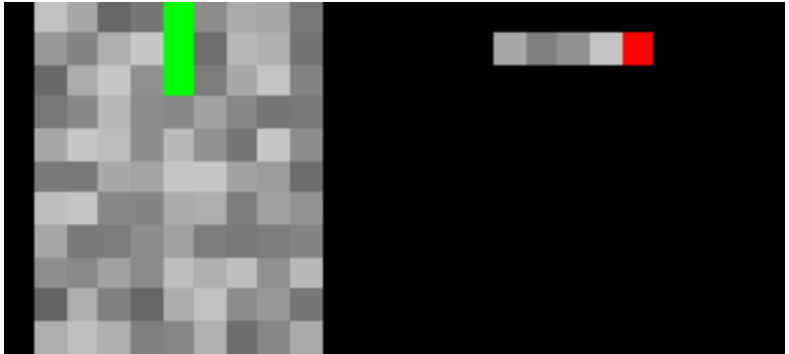
```
int in[W*H];
int bx[W*H];
int by[W*H];
for(int y=0; y<H; y++){
  for(int x=1; x<(W-1); x++{
    bx[x + y*W] = in[x-1 + y*W] + in[x + y*W] + in[x+1 +y*W];
  }
}
for(int y=1; y<(H-1); y++){
  for(int x=1; x<(W-1); x++{
    by[x + y*W] = bx[x + (y-1)*W] + bx[x + y*W] + bx[x+ (y+1)*W];
  }
}
```

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

Blur Stored Implementation

C code

```
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 Fusion

C code

```
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];
  }
  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];
  }
}
```

- Not as easily parallelizable
- High locality
  - producer bx & consumer by moved close together
- No recomputation

Blur Fusion

C code

```
int in[W*H];
int bx[W*H];
int by[W*H];
for(int y=0; y<2; y++){        // calculate 2 lines of bx
  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++){         // repeat calculating
  for(int x=1; x<(W-1); x++{        //   1 line of bx and 1 line of by
    bx[x + (y+1)*W] = in[x-1 + (y+1)*W] + in[x + (y+1)*W] + in[x+1 +
(y+1)*W];
  }
  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];
  }
}
```

- Not as easily parallelizable
- High locality
  - producer bx & consumer by moved close together
- No recomputation

Blur Fusion

C code

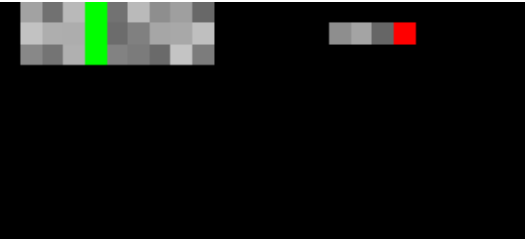
```
int in[W*H];
int bx[W*H];
int by[W*H];
for(int y=0; y<2; y++){
  for(int x=1; x<(W-1); x++{
    bx[x + y*W] = in[x-1 + y*W] + in[x + y*W] + in[x+1 + y*W];
  }
}
for(int y=1; y<(H-1); y++){
  for(int x=1; x<(W-1); x++{
    bx[x + (y+1)*W] = in[x-1 + (y+1)*W] + in[x + (y+1)*W] + in[x+1 + (y+1)*W];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

- Not as easily parallelizable
- High locality
  - producer bx & consumer by moved close together
- No recomputation

Blur Fusion – Folded Storage

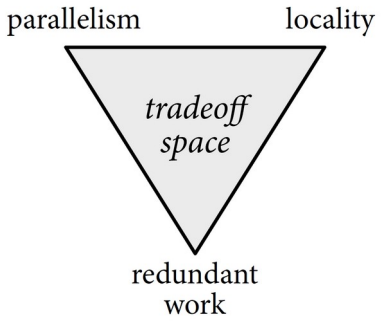
C code

```
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];
  }
  for(int x=1; x<(W-1); x++{
    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)

Some Remarks



- C allows us to very specifically program the execution order.
- Basically **trade-off space :**
- Loop fusion, storage folding can give us performance and storage size advantages.
  - (Many more possibilities: re-ordering, tiling, multithreading, vectorizing…)
    Most of them **obscure the functionality**.
    Most of them are **architecture specific**
    Rewriting and debugging to efficiently optimize
    High-level design-space exploration is discouraged by increasingly complex code.
- More stages ☾ **even more complex, unreadable code**

```
#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)];
  for (int xTile = 0; xTile < out.width(); xTile += 128) {
    __m128i *tmpPtr = tmp;
    for (int y = 0; y < 32+2; y++) {
      const uint16_t *inPtr = &(in(xTile, yTile+y));
      for (int x = 0; x < 128; x += 8) {
        a = _mm_load_si128((const __m128i*)(inPtr));
        b = _mm_loadu_si128((const __m128i*)(inPtr+1));
        c = _mm_loadu_si128((const __m128i*)(inPtr+2));
        sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
        avg = _mm_mulhi_epi16(sum,one_third);
        _mm_store_si128(tmpPtr++, avg);
        inPtr+=8;
      }
    }
    tmpPtr = tmp;
    for (int y = 0; y < 32; y++) {
      __m128i *outPtr = (__m128i *)(&(out(xTile, yTile+y)));
      for (int x = 0; x < 128; x += 8) {
        a = _mm_load_si128(tmpPtr+(2*128)/8);
        b = _mm_load_si128(tmpPtr+128/8);
        c = _mm_load_si128(tmpPtr++);
        sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
        avg = _mm_mulhi_epi16(sum, one_third);
        _mm_store_si128(outPtr++, avg);
      }
    }
  }
}
```

```
Func blur_3x3(Func in) {
  Func bx, by;
  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, 128, 32)
        .vectorize(xi, 8).parallel(y);
  bx.compute_at(by, x).vectorize(x, 8);

  return by;
}
```

Halide DSL and Compiler for Image Processing Pipelines

- Front-end embedded in C++
- Compiler can target many back-ends
  - Including x86/SSE, ARM v7/NEON, CUDA, Native Client, and OpenCL, WASM, The Qaulcomm Hexagon DSP
- Support from industry (Google, Adobe)
  - Google Pixel camera / Pixel Core

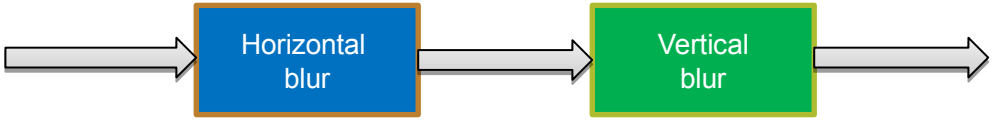
Main idea:

- Decouples algorithm definition from optimization schedule
  - Apply optimizations without complicating the code

Result:

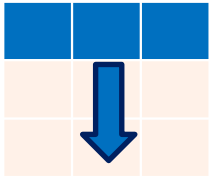
- Easier and faster design space exploration
- Improved code readability and portability
  - For a new architecture we should only change/rewrite the schedule

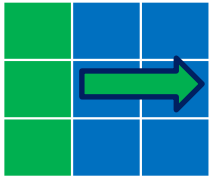
Blur Example Halide



$$bx(x,y)=in(x-1,y)+in(x,y)+in(x+1,y)$$

$$by(x,y)=bx(x,y-1)+bx(x,y)+bx(x,y+1)$$





```
Func in, bx, by;
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);    //≈ build and execute the loop nest
                          over a 10x10 area
```

Blur Example 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)

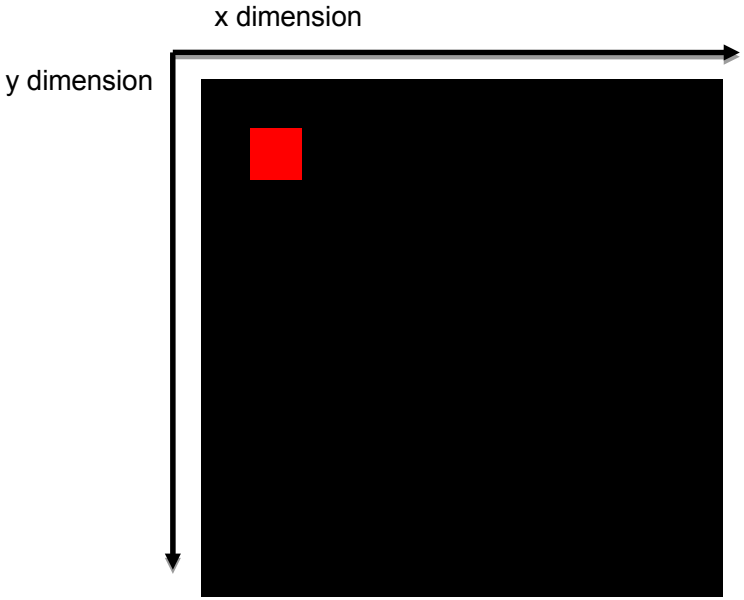
↓

- 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: Inter-stage scheduling - inline

```
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);
```



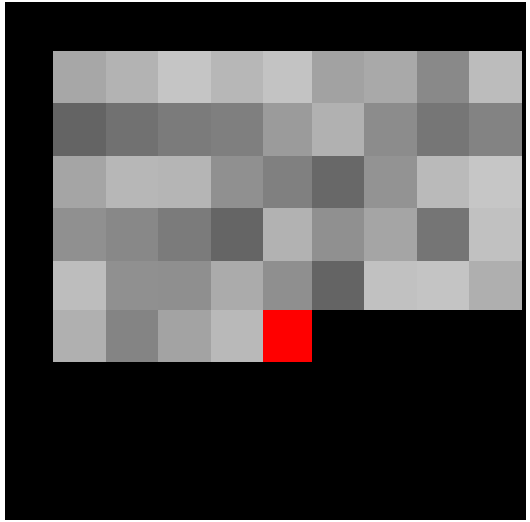
- Internally, Halide converts this functional representation to a C-like loop nest.
- By default, if nothing else is done, everything is **inlined.**

Scheduling: Inter-stage scheduling - inline

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);
```



C equivalent

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

Scheduling Inter-stage 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);
```

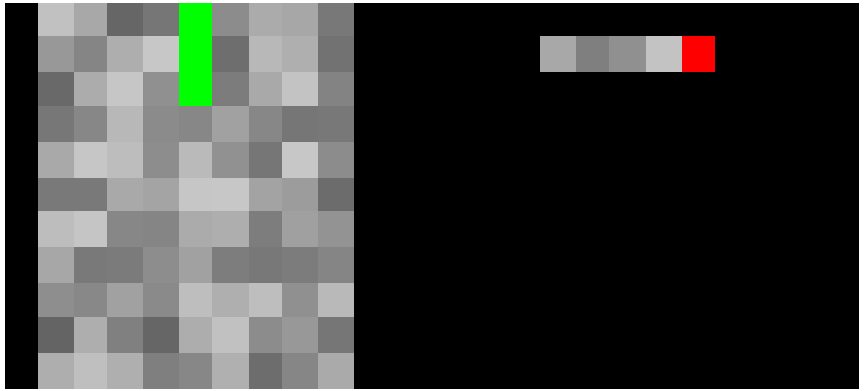
`compute_root()`: compute and store all outputs of a (producer) function before starting computation of the next

Scheduling Inter-stage 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);
```



C equivalent

```
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];
  }
}
```

Scheduling Fusion

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);
```

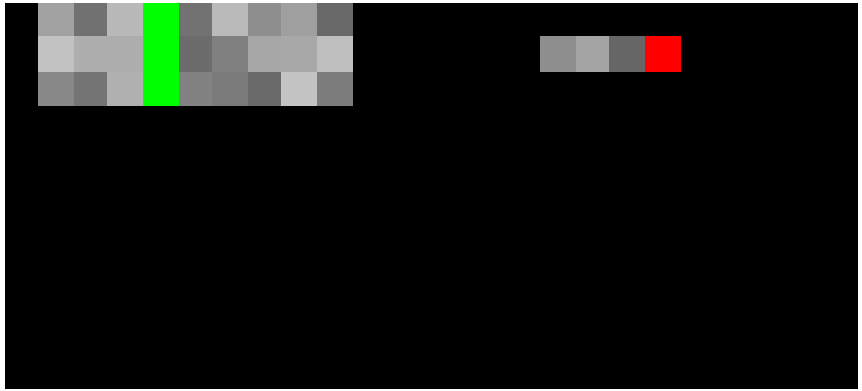
- **`compute_root()`** is actually a special case of **`compute_at()`**.
- **`bx.compute_at(by, y)`** means:
  "Whenever stage by starts an iteration of the y loop, first calculate those pixels of stage bx that will be consumed."
- In other words: computation of **bx is fused at the loop over y of by**.

Scheduling Fusion

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);
```



C equivalent

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

At this point in the nest:

- **Allocate a buffer** for bx
- Fill it with the required **3 lines**.

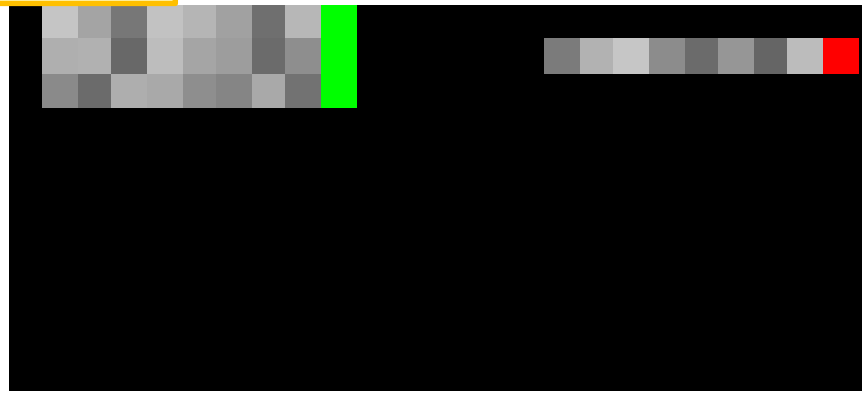
Scheduling Fu

This is not equivalent to our initial fused C implementation yet, because the bx pixels are **not being re-used but instead re-calculated**.

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);
```



C equivalent

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

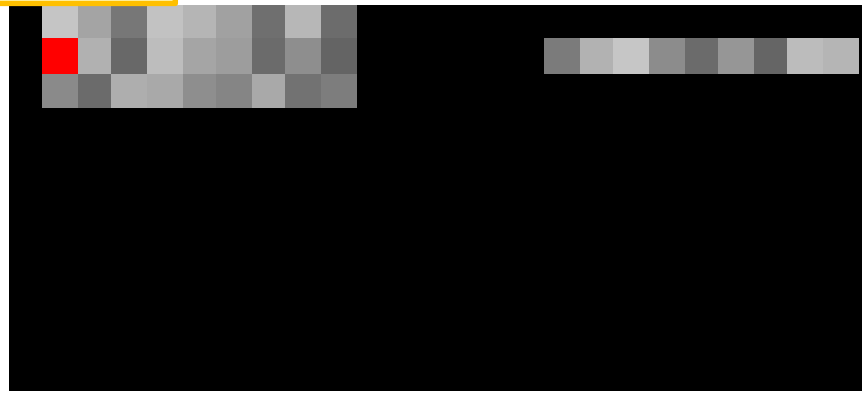
Scheduling Fusion

This is not equivalent to our initial fused C implementation yet, because the bx pixels are **not being re-used but instead re-calculated**.

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);
```



C equivalent

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

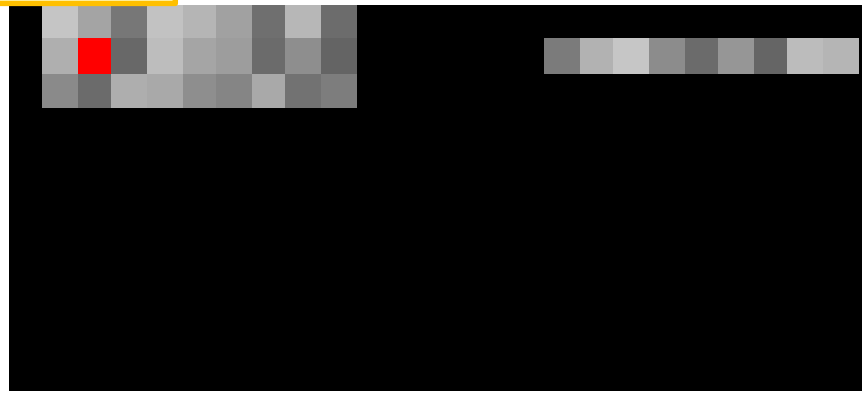
Scheduling Fusion

This is not equivalent to our initial fused C implementation yet, because the bx pixels are **not being re-used but instead re-calculated**.

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);
```



C equivalent

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

Scheduling Fusion-Folded storage

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);
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."*

Scheduling Fusion-Folded Storage

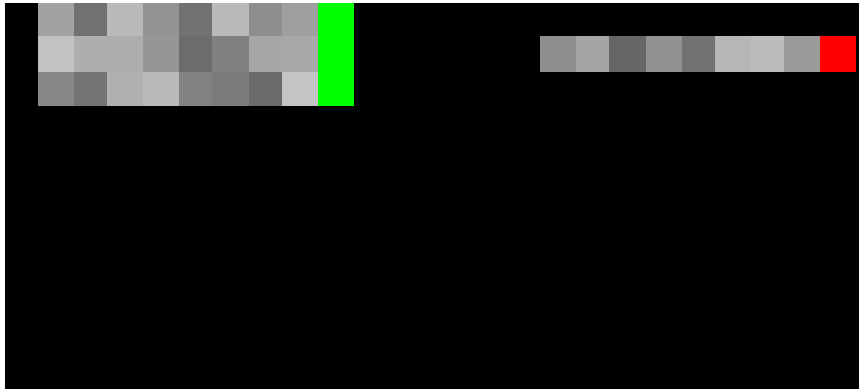
Halide

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

C equivalent

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



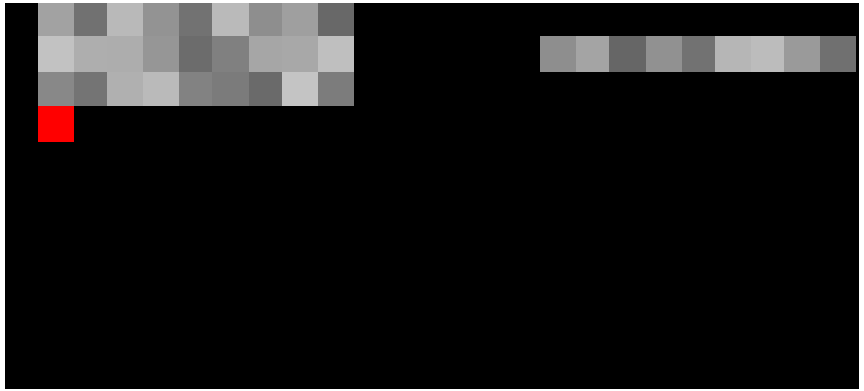
Scheduling Fusion-Folded Storage

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

C equivalent



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

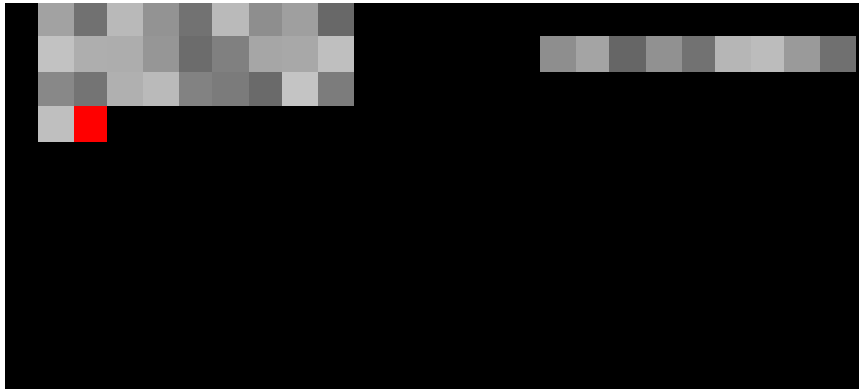
Scheduling Fusion-Folded Storage

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

C equivalent



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

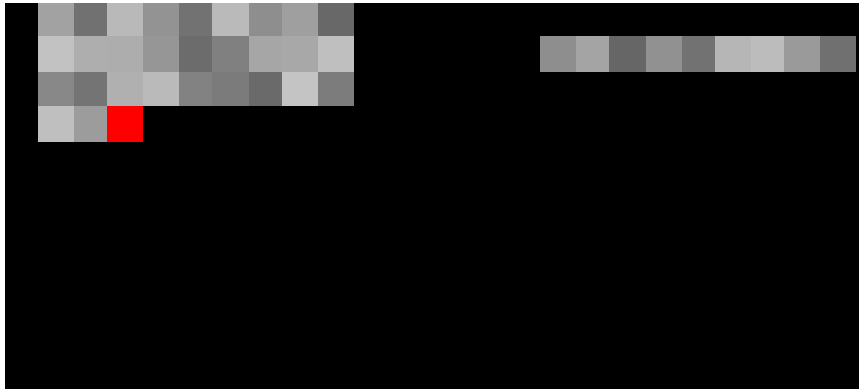
Scheduling Fusion-Folded Storage

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

C equivalent



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

Halide automatically applies the storage folding as well!

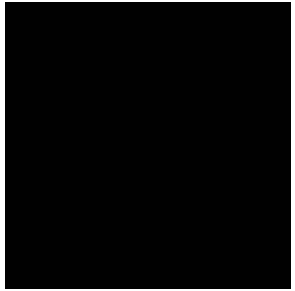
Intra-Stage Scheduling

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

Intra-Stage Scheduling

- We looked at the syntax which **interleaves computation between stages**.
- There is also syntax which changes the **order of computation within a single stage**:

```
gradient.realize(4,4);
     Default - >
```



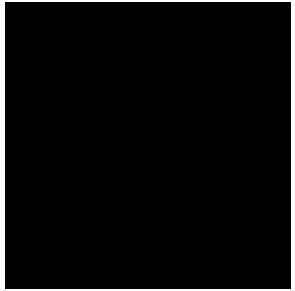
```
Func gradient;
 gradient(x,y)=x+y;
```

Intra-Stage Scheduling: Loop Interchange

- 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 ☾ **gradient.reorder(y,x);**

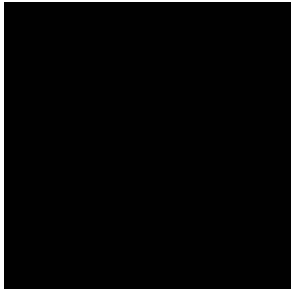
```
gradient.realize(4,4);
```

Default - >



```
gradient.reorder(y,x);
gradient.realize(4,4);
```

Loop interchange - >



```
Func gradient;
gradient(x,y)=x+y;
```

Intra-Stage Scheduling: Splitting

- 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
    - **gradient.reorder(y,x);**
  - **Split** loop variables into inner and outer
    - **gradient.split(x, xout, xin, 4);**
  - **Tiling** is a just combination of the above:
    - **gradient.split(x, xout, xin, 4);**
    - **gradient.split(y, yout, yin, 4);**
    - **gradient.reorder(xin, yin, xout, yout);**

```
Func gradient;
gradient(x,y)=x+y;
```

Intra-Stage Scheduling: Tiling

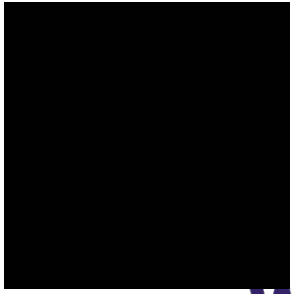
- We looked at the syntax which **interleaves computation between stages**.

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

```
Func gradient;
gradient(x,y)=x+y;
```

```
gradient.realize(8,8);
```

Loop tiling - >



Intra-Stage Scheduling: Vectorize

- 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 `gradient.reorder(y,x);`
  - **Split** loop variables into inner and outer `gradient.split(x, xout, xin, 4);`
  - **Tiling** is a just combination of the above `gradient.tile(x, y, xout, yout , xin, yin, 4, 4);`

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

```
gradient.vectorize(x,4)
gradient.realize(8, 4);
```

```
Func gradient;
gradient(x,y)=x+y;
```

Vectorization - >

Intra-Stage Scheduling

- 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
    ```
    gradient.reorder(y,x);
    ```
  - **Split** loop variables into inner and outer
    ```
    gradient.split(x, xout, xin, 4);
    ```
  - **Tiling** is a just combination of the above
    ```
    gradient.tile(x, y, xout, yout ,
    xin, yin, 4, 4);
    ```
  - **Vectorize** loop iterations
    ```
    gradient.vectorize(x, 4);
    ```

```
Func gradient;
gradient(x,y)=x+y;
```

Intra-Stage Scheduling: Multiple Threads

- 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 `gradient.reorder(y,x);`
  - **Split** loop variables into inner and outer `gradient.split(x, xout, xin, 4);`
  - **Tiling** is a just combination of the above `gradient.tile(x, y, xout, yout , xin, yin, 4, 4);`
  - **Vectorize** loop iterations `gradient.vectorize(x, 4);`

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

```
Func gradient;
gradient(x,y)=x+y;
```

Intra-Stage Scheduling

- 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 `gradient.reorder(y,x);`
  - **Split** loop variables into inner and outer `gradient.split(x, xout, xin, 4);`
  - **Tiling** is a just combination of the above `gradient.tile(x, y, xout, yout , xin, yin, 4, 4);`
  - **Vectorize** loop iterations `gradient.vectorize(x, 4);`
  - **Parallelize** loop iterations `gradient.parallel(x);`
  - **Many more!** (unrolling, merging, prefetching, storage ordering etc)

```
Func gradient;
gradient(x,y)=x+y;
```

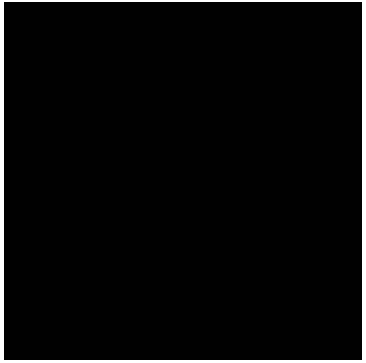
Intra-Stage Scheduling

- 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 ☾ `gradient.reorder(y,x);`
  - **Split** loop variables into inner and outer ☾ `gradient.split(x, xout, xin, 4);`
  - **Tiling** is a just combination of the above ☾ **gradient.tile(x, y, xout, yout , xin, yin, 4, 4);**
  - **Vectorize** loop iterations ☾ `gradient.vectorize(x, 4);`
  - **Parallelize** loop iterations ☾ `gradient.parallel(x);`
  - **Many more!** (unrolling, merging, prefetching, storage ordering etc)

And of course combinations of the above!

```
gradient.tile(x, y, xout, yout, xin, yin, 2, 2)
.fuse(xout, yout, tile_index)
.parallel(tile_index);
gradient.realize(8, 8);
```

Parallel tiles - >



```
Func gradient;
gradient(x,y)=x+y;
```

GPU Example

```
Func in, bx, by;
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);
```

We start with the same functional definitions

But first: a short GPU / CUDA recap

GPU Recap: Let's Start Again from C

```
int A[2][4];
for(i=0;i<2;i++)
    for(j=0;j<4;j++)
        A[i][j]++;
```

convert into CUDA

```
int A[2][4];
kernelF<<<(2,1),(4,1)>>>(A);
__device__    kernelF(A){
    i = blockIdx.x;
    j = threadIdx.x;
    A[i][j]++;
}
```

// define 2x4=8 threads

// all threads run the same kernel

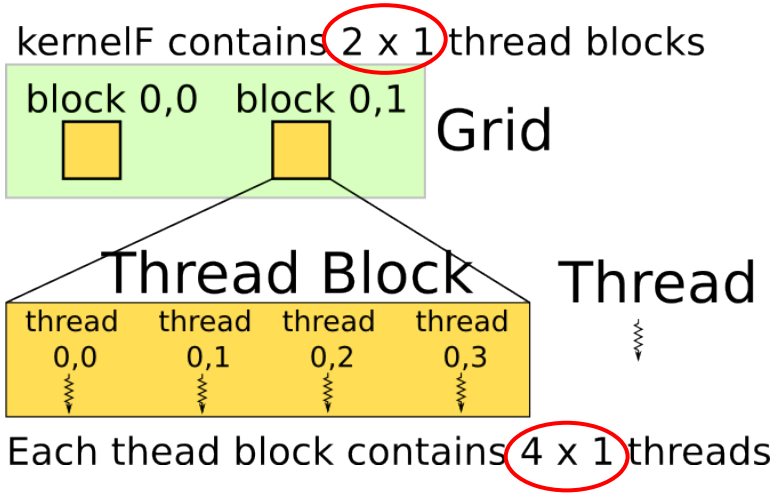
// each thread block has its id

// each thread has its id

// each thread has different i and j

Thread Hierarchy

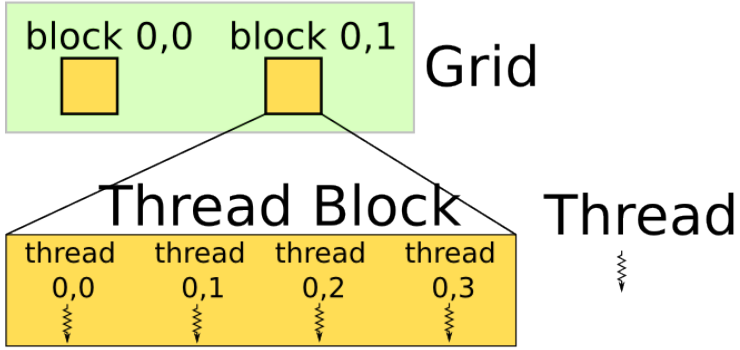
Example:
thread 3 of block 1 operates
on element A[1][3]



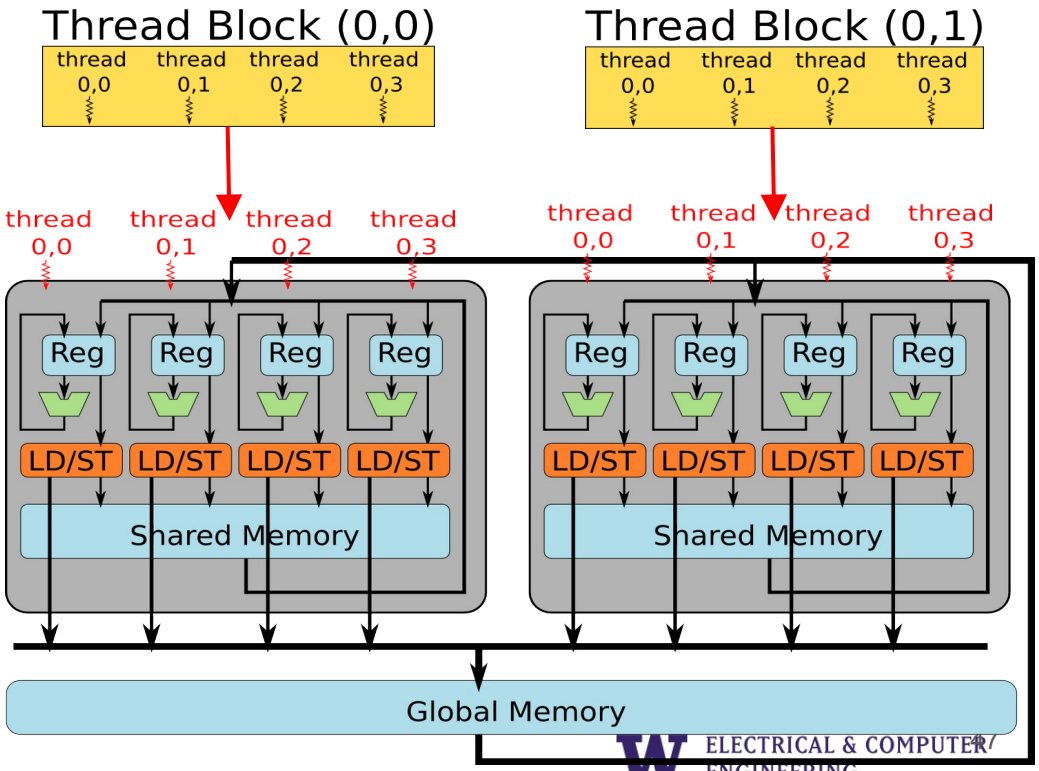

```
int A[2][4];
kernelF<<<(2,1),(4,1)>>>(A);  // define 2x4=8 threads
__device__    kernelF(A){     // all threads run same kernel
    i = blockIdx.x;           // each thread block has its id
    j = threadIdx.x;          // each thread has its id
    A[i][j]++;                // each thread has different i and j
}
```

How Are Threads Scheduled?

kernelF contains 2 x 1 thread blocks



Each thead block contains 4 x 1 threads



GPU example

```
Func in, bx, by;
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);
```

Now let's compute the bx stage using the **GPU** in 16-wide one-dimensional **thread blocks**

```
Var block, thread;
bx.split(x,block,thread,16);
```

1. First we **split** the index x of bx into blocks of size 16

GPU Example

```
Func in, bx, by;
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);
```

Now let's compute the bx stage using the **GPU** in 16-wide one-dimensional **thread blocks**

```
Var block, thread;
bx.split(x,block,thread,16);
```

1. First we **split** the index x of bx into blocks of size 16

```
bx.gpu_blocks(block)
  .gpu_threads(threads);
```

2. Then we tell cuda that our Vars 'block' and 'thread' correspond to **CUDA**'s notions of **blocks and threads**, or OpenCL's notions of thread groups and threads.

```
bx.gpu_tile(x, block, thread, 16);    // short-hand notation
```

GPU Example

```
Func in, bx, by;
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);
```

We can also combine these with all of the previous inter-stage scheduling

GPU Example: A Better Schedule

```
Func in, bx, by;
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);

Var xo, yo, xin, yin;
by.compute_root().
  .gpu_tile(x,y,xo,yo,
        xin,yin,16,16);
```

1. by (output stage) is set to **root**
2. by tiled with **tiles of size 16** (x,y dimensions)

GPU Example: A Better Schedule

```
Func in, bx, by;
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);

Var xo, yo, xin, yin;
by.compute_root().
  .gpu_tile(x,y,xo,yo,
        xin,yin,16,16);
```

Remember that root allocations mean large buffers!!
(stored in the global memory)

1. by (output stage) is set to **root**
2. by tiled with **tiles of size 16** (x,y dimensions)

GPU Example
(a better schedule)

```
Func in, bx, by;
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);

Var xo, yo, xin, yin;
by.compute_root().
  .gpu_tile(x,y,xo,yo,
        xin,yin,16,16);


bx.compute_at(by,xo)
```

1. by (output stage) is set to **root**
2. by tiled with **tiles of size 16** (x,y dimensions)
3. Produce bx **inside tiles (blocks)** of
   bx in the **shared (on-chip)**
   ry

Increased producer consumer locality reduces the number of costly global memory accesses!!!

GPU Example
(a better schedule)

```
Func in, bx, by;
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);

Var xo, yo, xin, yin;
by.compute_root().
  .gpu_tile(x,y,xo,yo,
        xin,yin,16,16);


bx.compute_at(by,xo)
  .gpu_threads(x,y);
```

1. by (output stage) is set to **root**
2. by tiled with **tiles of size 16** (x,y dimensions)
3. Produce bx **inside tiles (blocks)** of **by**
   Store bx in the **shared (on-chip) memory**
   Assign x,y dimensions of bx to **threads**

GPU Example
(a better schedule)

```
Func in, bx, by;
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);

Var xo, yo, xin, yin;
by.compute_root().
   .gpu_tile(x,y,xo,yo,
         xin,yin,16,16);


bx.compute_at(by,xo)
   .gpu_threads(x,y);
```

1. by (output stage) is set to **root**
2. by tiled with **tiles of size 16** (x,y dimensions)
3. Produce bx **inside tiles (blocks)** of

   in the **shared (on-chip)**

But tile sizes control the shared memory requirements for bx and therefore the shared memory allocation size and thread block dimensions!!

GPU Example - compute per block

(common GPU fusion strategy)

$$by(x,y)=bx(x,y)+bx(x,y+1)+bx(x,y+2)$$

```
bx.compute_at(by,xo)
  .gpu_threads(x,y);
```

```
by.compute_root().gpu_tile(x,y,xo,yo,
                           xin,yin,4,4);
```

Ty= 4

Tx= 4

GPU Example - compute per block

(common GPU fusion strategy)

$$by(x,y)=bx(x,y)+bx(x,y+1)+bx(x,y+2)$$

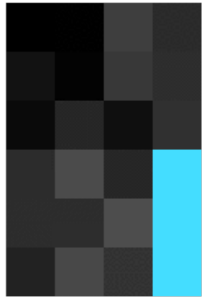
```
bx.compute_at(by,xo)
  .gpu_threads(x,y);
```

```
by.compute_root().gpu_tile(x,y,xo,yo,
                           xin,yin,4,4);
```

Ty= 4

Tx= 4

blur_x



blur_y

GPU Example - compute per block

(common GPU fusion strategy)

$$by(x,y)=bx(x,y)+bx(x,y+1)+bx(x,y+2)$$

```
bx.compute_at(by,xo)
   .gpu_threads(x,y);
```

```
by.compute_root().gpu_tile(x,y,xo,yo,
                           xin,yin,4,4);
```

Ty= 4

Tx= 4

Tiling by causes generates a thread block of 4x4 (root allocations are stored in the global memory)

GPU Example - compute per block

(common GPU fusion strategy)

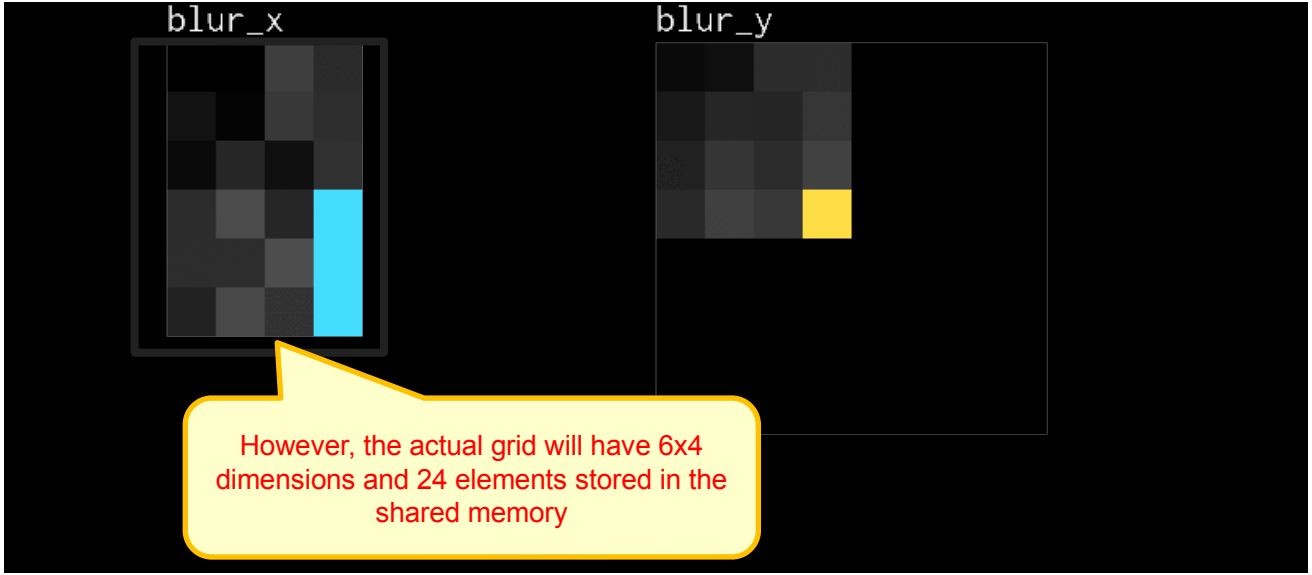
$$by(x,y)=bx(x,y)+bx(x,y+1)+bx(x,y+2)$$

```
bx.compute_at(by,xo)
   .gpu_threads(x,y);
```

```
by.compute_root().gpu_tile(x,y,xo,yo,
                           xin,yin,4,4);
```

Ty= 4

Tx= 4



GPU example - compute per block

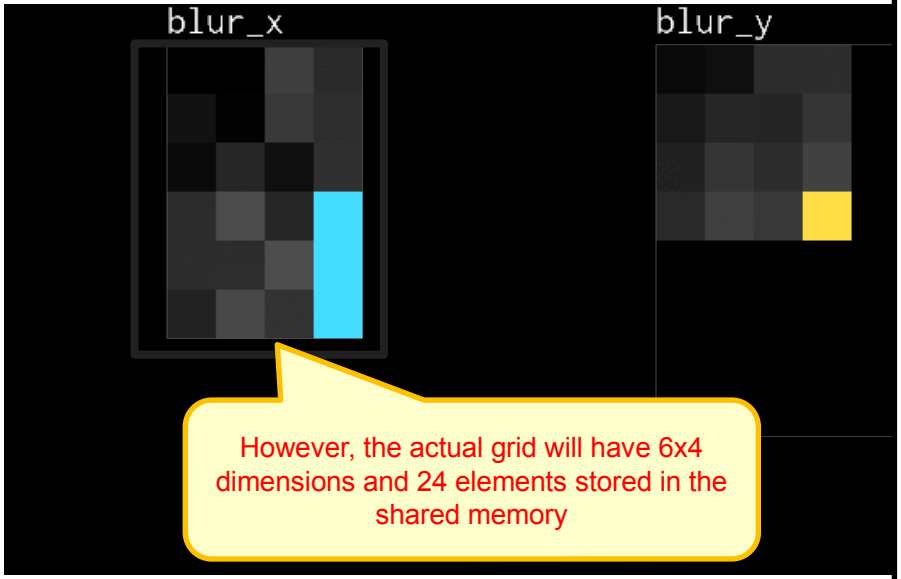
(common GPU fusion strategy)

$$by(x,y)=bx(x,y)+bx(x,y+1)+bx(x,y+2)$$

```
bx.compute_at(by,xo)
   .gpu_threads(x,y);
```

```
by.compute_root().gpu_tile(x,y,xo,yo,
                           xin,yin,4,4);
```

Ty=
4
Ty=



However, the actual grid will have 6x4 dimensions and 24 elements stored in the shared memory

GPUs have specific constraints for following metrics!

Max threads/block
(1024)
Max shared memory/block
(49 Kbytes)
Max registers per SM (Streaming Multi-processor) varies per CC (Compute Capability)
etc…

GPU Example - compute per block

(common GPU fusion strategy)

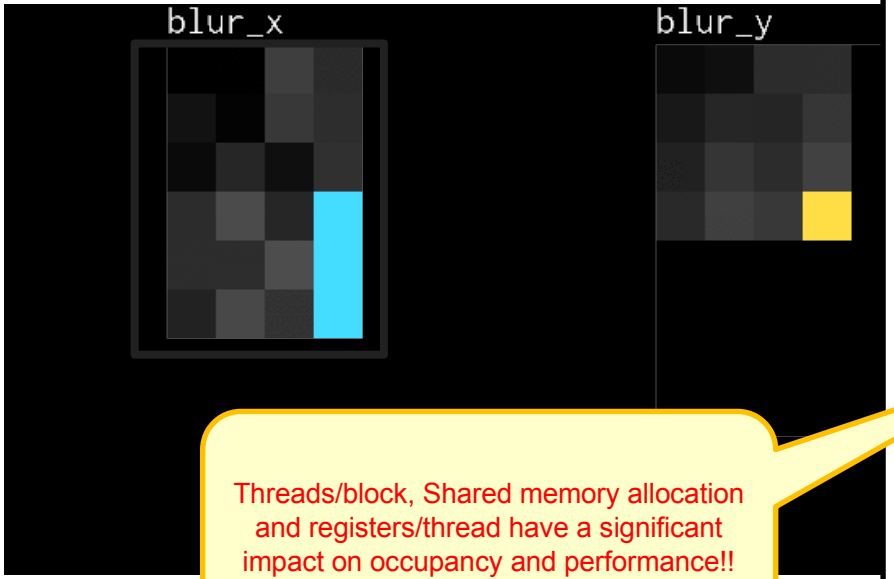
$$by(x,y)=bx(x,y)+bx(x,y+1)+bx(x,y+2)$$

```
bx.compute_at(by,xo)
  .gpu_threads(x,y);
```

```
by.compute_root().gpu_tile(x,y,xo,yo,
                           xin,yin,4,4);
```

Ty= 4

Tx=



Threads/block, Shared memory allocation and registers/thread have a significant impact on occupancy and performance!!

GPUs have specific constraints for these following metrics!

Max threads/block (1024)
Max shared memory/block (49 Kbytes)
Max registers per SM (varies per CC)
etc…

GPU Example - compute per block

(common GPU fusion strategy)

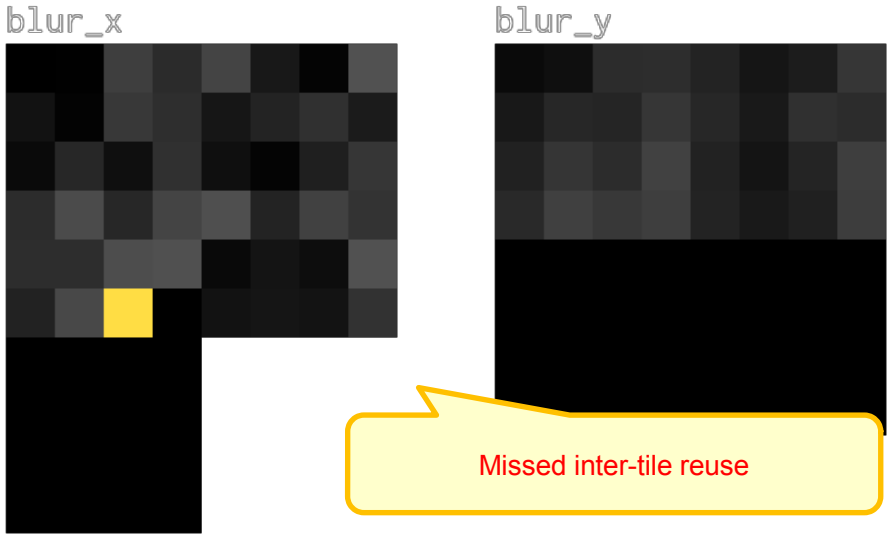
$$by(x,y)=bx(x,y)+bx(x,y+1)+bx(x,y+2)$$

```
bx.compute_at(by,xo)
  .gpu_threads(x,y);
```

```
by.compute_root().gpu_tile(x,y,xo,yo,
                           xin,yin,4,4);
```

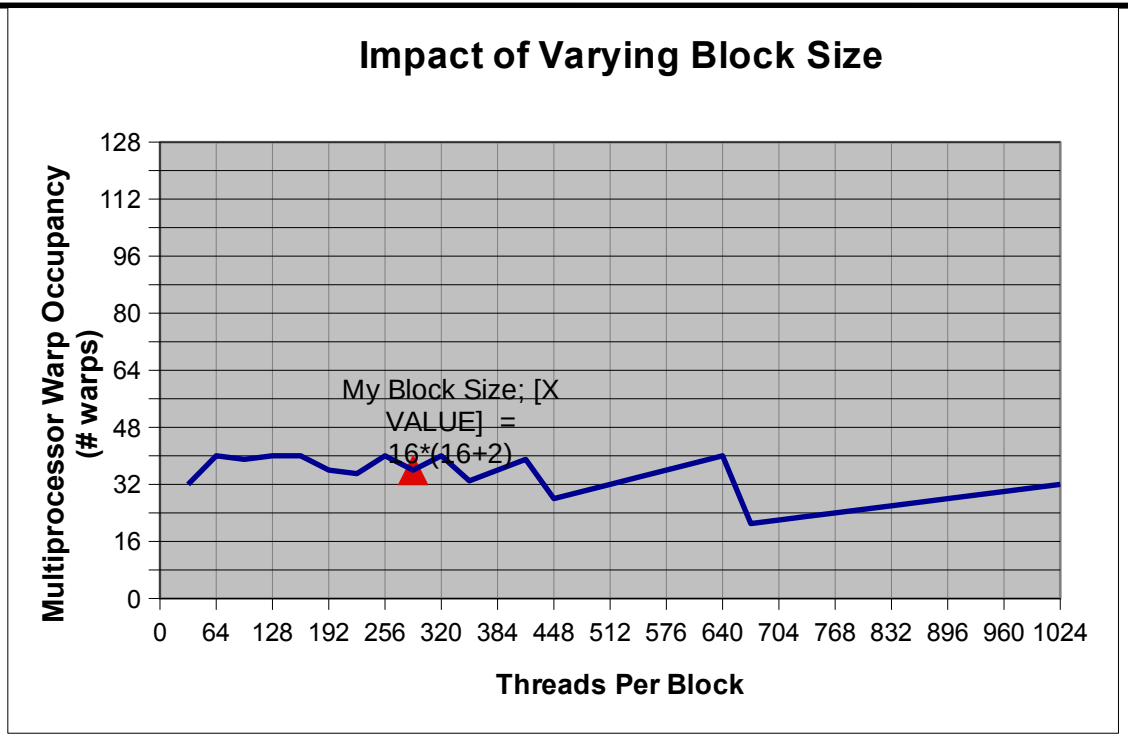
Ty= 4

Tx= 4

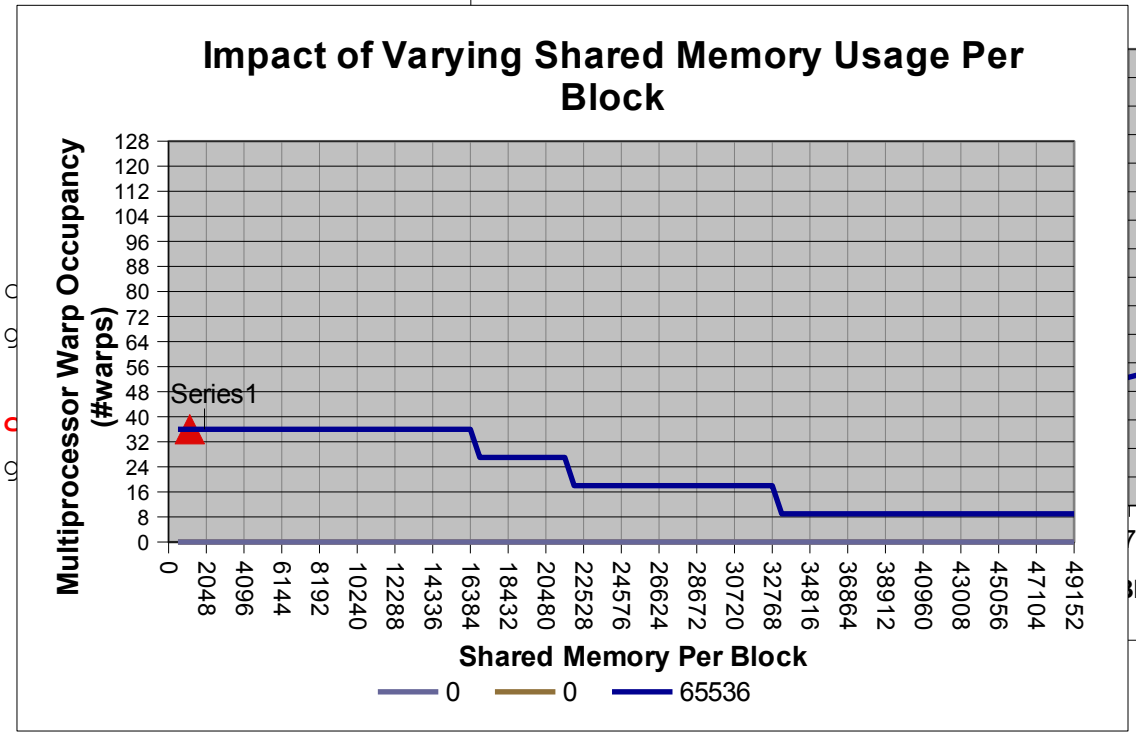


Occupancy

```
by.compute_root().
  .gpu_tile(x,y,xo,yo,
        xin,yin,16,16);
bx.compute_at(by,xo)
  .gpu_threads(x,y);
```



Occupancy


Impact of Varying Shared Memory Usage Per Block
Multiprocessor Warp Occupancy (#warps)
128
120
112
104
96
88
80
72
64
56
48
40
32
24
16
8
0
Series1
0
2048
4096
6144
8192
10240
12288
14336
16384
18432
20480
22528
24576
26624
28672
30720
32768
34816
36864
38912
40960
43008
45056
47104
49152
Shared Memory Per Block
0
0
65536

Impact of Varying Block Size
76 640 704 768 832 896 960 1024
Block

Observations

- With Halide, the algorithm definition is **clearer and conciser** than with C.
  - by being separated from the optimization (scheduling & buffer allocation) 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** (pro-, epilogues) and **storage folding**, among other optimizations

Limitations

- As mentioned it is **domain-specific** to image processing. It is 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 get something like C again!**

Halide's Status

- An open-source project.
- Because of that, there is ongoing work in many directions:
  - **New architectures** (i.e. Qualcomm Hexagon DSP)
  - Extensions to suit (sub-)domains (i.e. **HLS** code generation)
  - Auxiliary efforts:
    - **Automatic tuning** of Halide schedules
    - **Automatic generation** of Halide schedules
    - **Automatic de-compilation of x86 binaries into Halide** for easy porting

GPU example - compute per block

(common GPU fusion strategy)

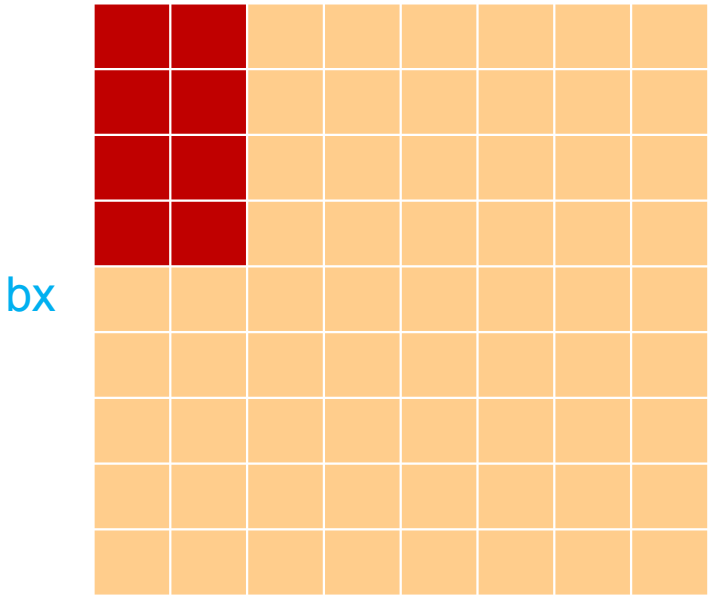
$$by(x,y)=bx(x,y)+bx(x,y+1)+bx(x,y+2)$$

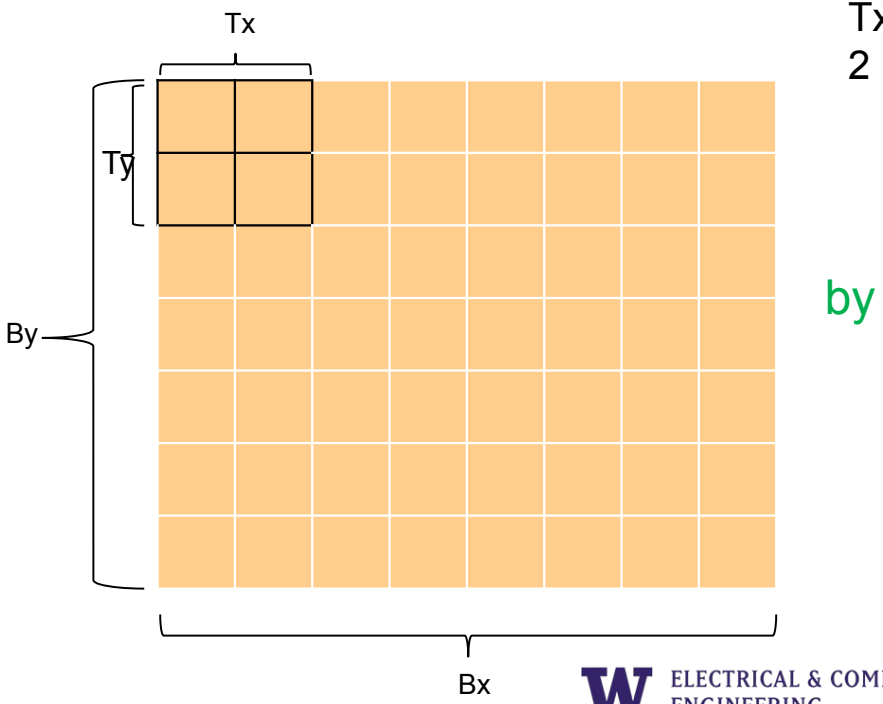
```
bx.compute_at(by,xo)
  .gpu_threads(x,y);
```

```
by.compute_root().gpu_tile(x,y,xo,yo,
                           xin,yin,2,2);
```

Ty= 2

Tx= 2





GPU Example - compute per block

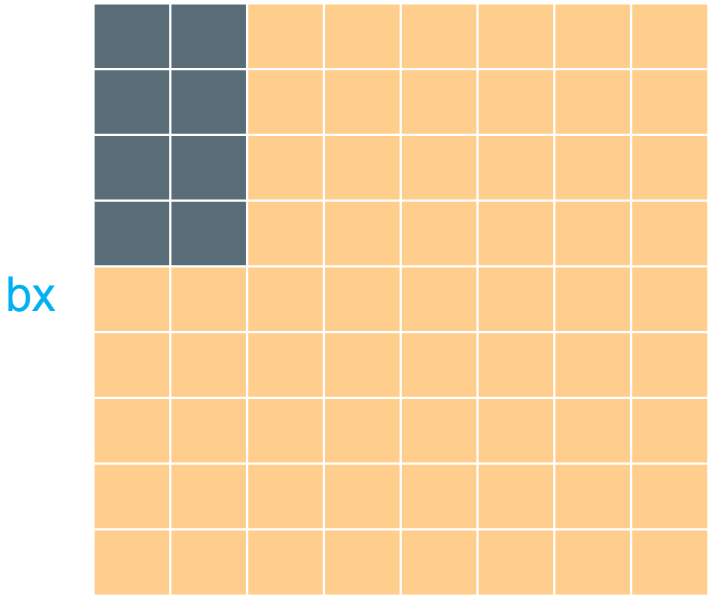
(common GPU fusion strategy)

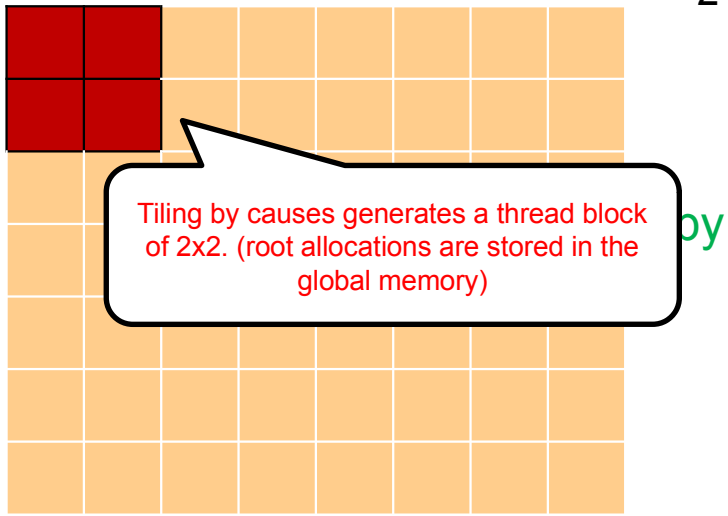
$$by(x,y)=bx(x,y)+bx(x,y+1)+bx(x,y+2)$$

```
bx.compute_at(by,xo)
  .gpu_threads(x,y);
```

```
by.compute_root().gpu_tile(x,y,xo,yo,
                           xin,yin,2,2);
```

Ty= 2
Tx= 2





GPU Example - compute per block

(common GPU fusion strategy)

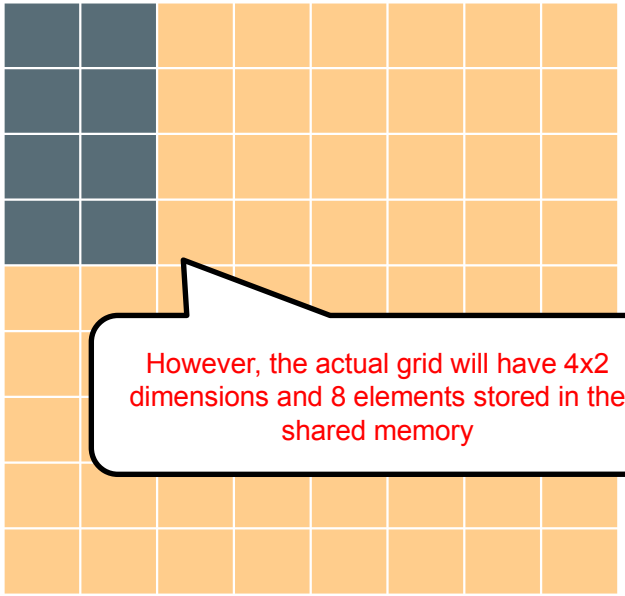
$$by(x,y)=bx(x,y)+bx(x,y+1)+bx(x,y+2)$$

```
bx.compute_at(by,xo)
  .gpu_threads(x,y);
```

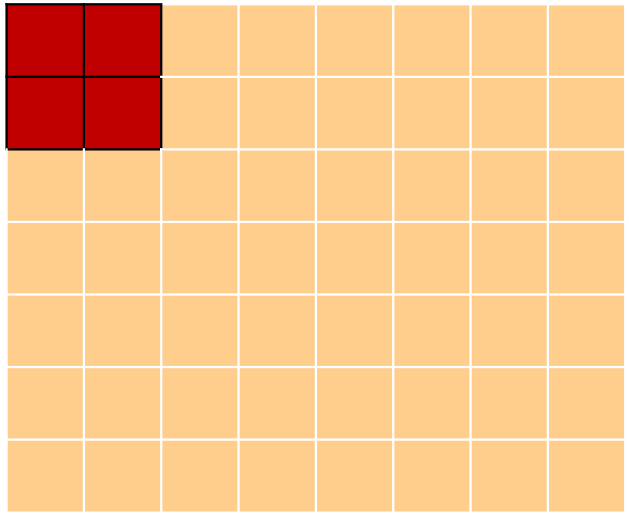
```
by.compute_root().gpu_tile(x,y,xo,yo,
                           xin,yin,2,2);
```

Ty= 2
Tx= 2

bx



However, the actual grid will have 4x2 dimensions and 8 elements stored in the shared memory



by

GPU example - compute per block

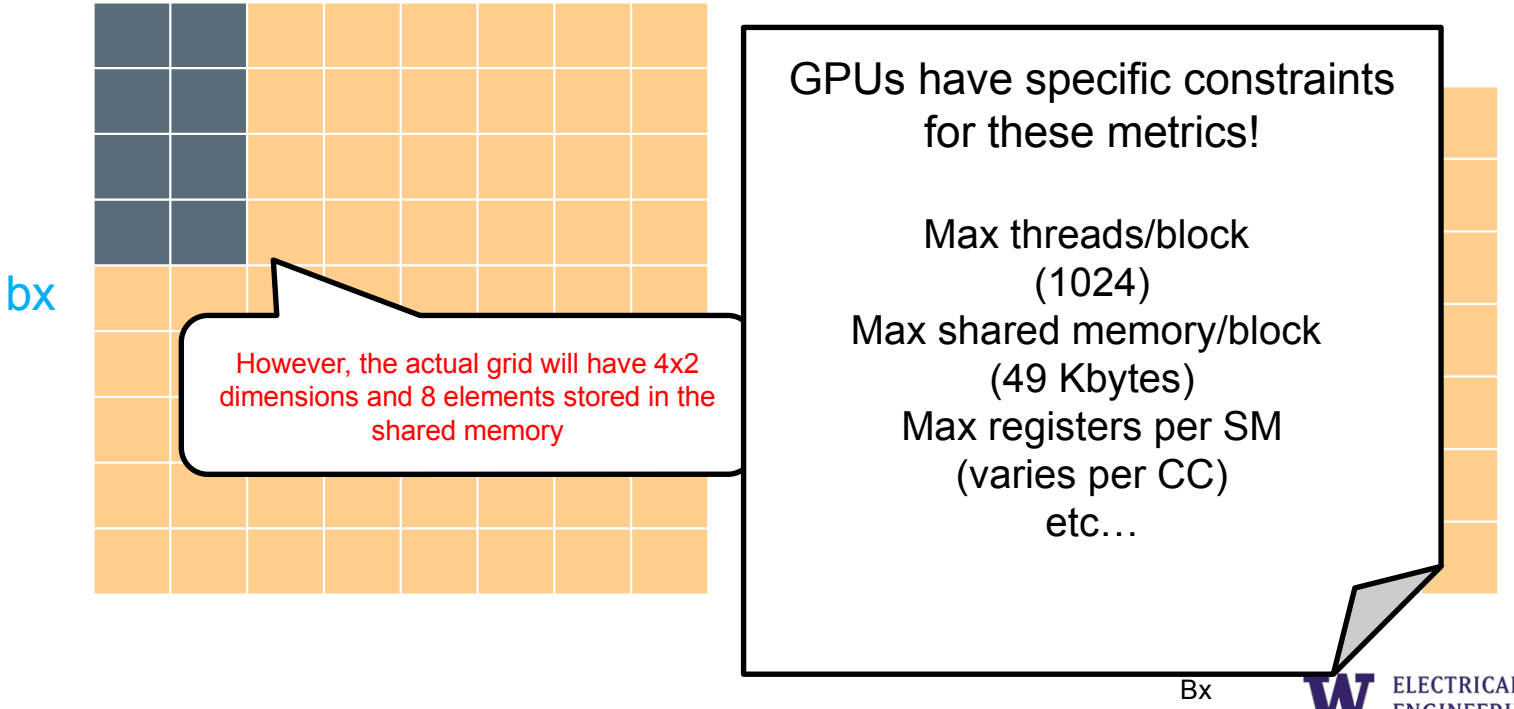
(common GPU fusion strategy)

```
bx.compute_at(by,xo)
  .gpu_threads(x,y);
```

$$by(x,y)=bx(x,y)+bx(x,y+1)+bx(x,y+2)$$

```
by.compute_root().gpu_tile(x,y,xo,yo,
                           xin,yin,2,2);
```

Ty= 2
Tx= 2



However, the actual grid will have 4x2 dimensions and 8 elements stored in the shared memory

GPUs have specific constraints for these metrics!

Max threads/block (1024)
Max shared memory/block (49 Kbytes)
Max registers per SM (varies per CC)
etc…

GPU Example - compute per block

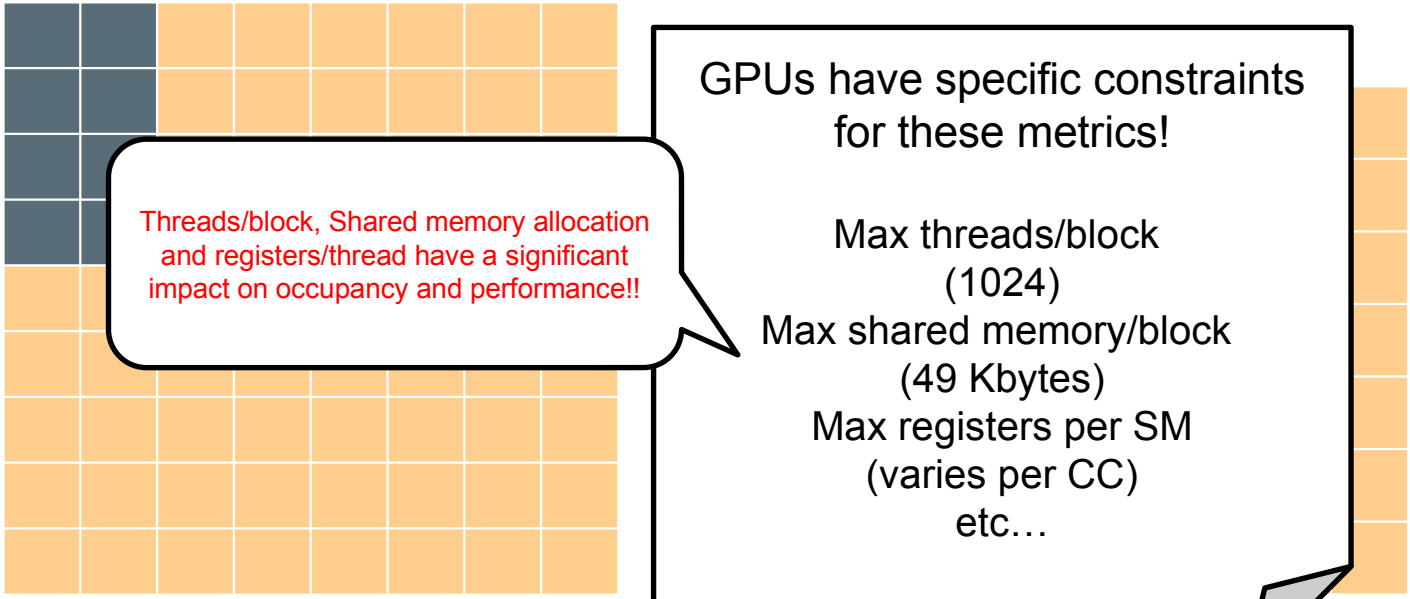
(common GPU fusion strategy)

$$by(x,y)=bx(x,y)+bx(x,y+1)+bx(x,y+2)$$

```
bx.compute_at(by,xo)
  .gpu_threads(x,y);
```

```
by.compute_root().gpu_tile(x,y,xo,yo,
                           xin,yin,2,2);
```

Ty= 2
Tx= 2



Threads/block, Shared memory allocation and registers/thread have a significant impact on occupancy and performance!!

GPUs have specific constraints for these metrics!

Max threads/block (1024)
Max shared memory/block (49 Kbytes)
Max registers per SM (varies per CC)
etc…