POLITECNICO DI MILANO

# Heterogeneous computing: Domain-Specific Languages and High-Level Synthesis

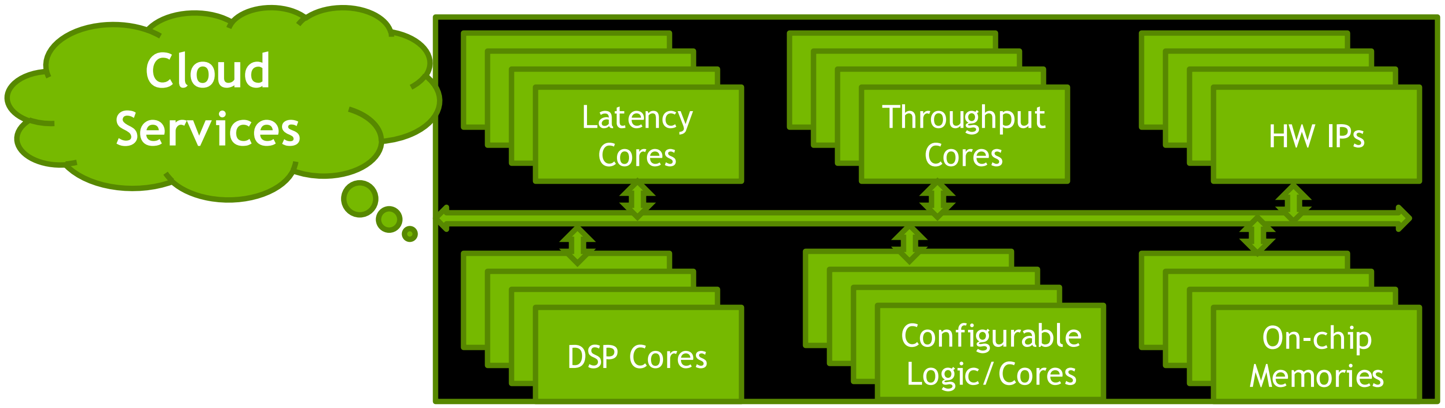
Parallel Computing

**Serena Curzel**

Politecnico di Milano
Dipartimento di Elettronica , Informazione e Bioingegneria
*serena.curzel@polimi.it*

## Heterogeneous processing

- ❑ Observation: most "real world" applications have complex workload characteristics
  - Parallelizable & hard to parallelize
  - SIMD patterns & divergent control flows
  - Predictable & unpredictable data accesses
- ❑ The best system is a heterogeneous one



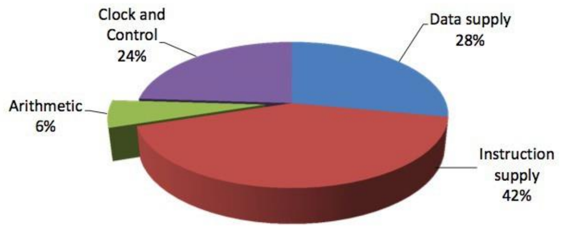
## Energy-efficient computing

- Given a fixed power budget, the goal is not only to increase performance, but also *energy efficiency*
- Specialization (i.e., fixed function hardware) can provide better energy efficiency

$$\boldsymbol{Power} = \frac{\boldsymbol{Op}}{\boldsymbol{second}} \times \frac{\boldsymbol{Joules}}{\boldsymbol{Op}}$$

## Energy-efficient computing

- General-purpose processors are not energy-efficient
- For each instruction:
  - Read instruction (Address translation, communicate with icache, access icache, etc.)
  - Decode instruction (Translate op to uops, access uop cache, etc.)
  - Check for dependencies/pipeline hazards
  - Identify available execution resource
  - Use decoded operands to control register file SRAM (retrieve data)
  - Move data from register file to selected execution resource
  - Perform arithmetic operation
  - Move data from execution resource to register file
  - Use decoded operands to control write to register file SRAM

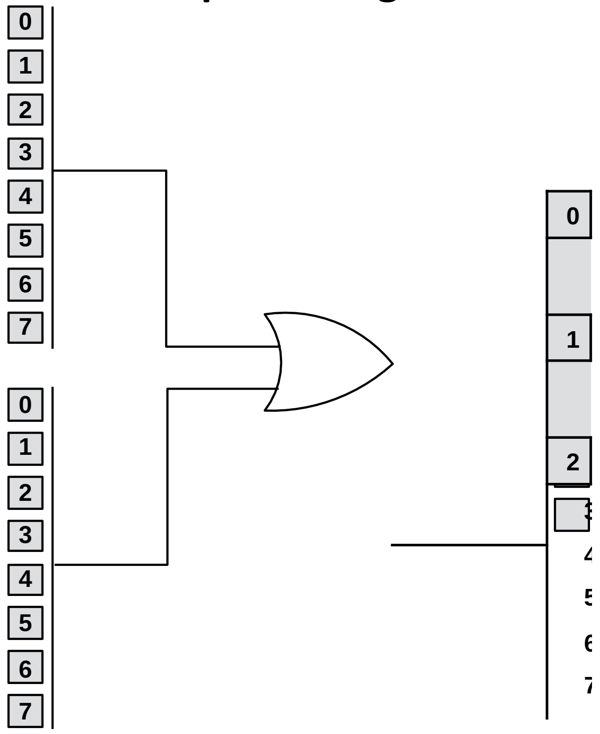


*Efficient Embedded Computing [Dally et al. 08]*

## Energy-efficient computing

- On the other hand, the circuit that would be required to actually perform the computation can be very simple

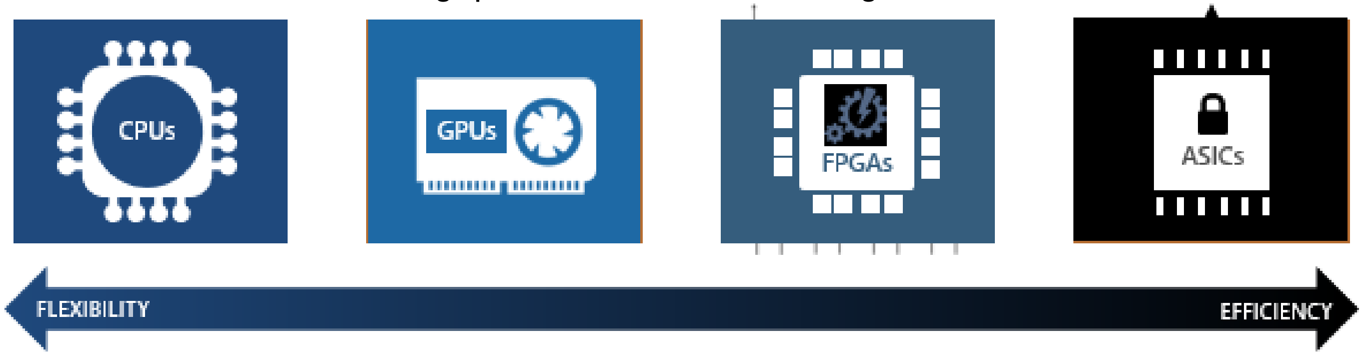
Example: 8-bit logical OR



# Heterogeneous processing

- Choosing the right tool for the job:

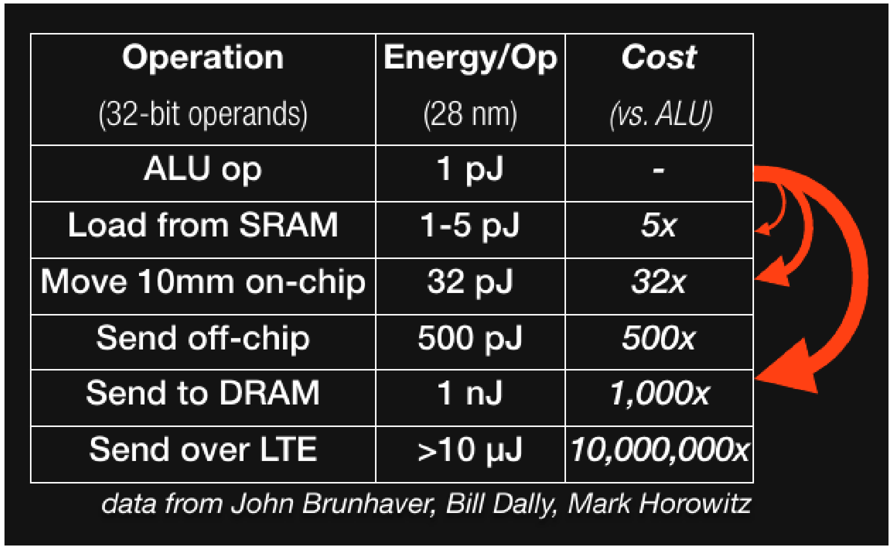
| Control-oriented | Throughput-oriented | Reconfigurable | Fixed function |
|---|---|---|---|



| | | | |
|---|---|---|---|
| | x10 more efficient | x100 more efficient? | x100-1000 more efficient |
| Easiest to program | | Difficult to program (active research on how to make it easier) | Not programmable<br>Costs 10-100M $ to design/verify/ manufacture |

## Energy-efficient computing

- Two ideas to reduce energy consumption:
  - Use specialized processors (more operations per joule, higher energy efficiency)
  - Move less data!
- Minimizing communication overhead increases performance, and it also reduces energy consumption

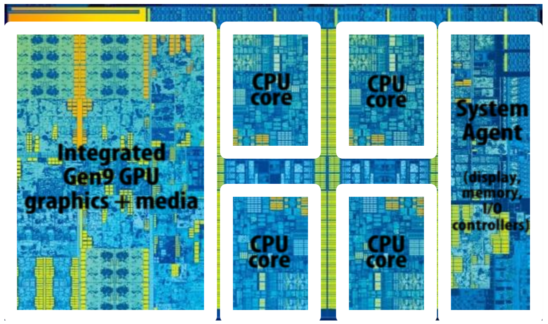


| **Operation** (32-bit operands) | **Energy/Op** (28 nm) | ***Cost*** *(vs. ALU)* |
|---|---|---|
| ALU op | 1 pJ | - |
| Load from SRAM | 1-5 pJ | *5x* |
| Move 10mm on-chip | 32 pJ | *32x* |
| Send off-chip | 500 pJ | *500x* |
| Send to DRAM | 1 nJ | *1,000x* |
| Send over LTE | >10 µJ | *10,000,000x* |

*data from John Brunhaver, Bill Dally, Mark Horowitz*

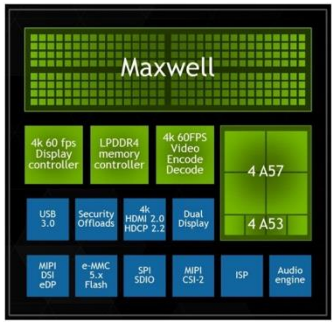
## Heterogeneous parallel programming

- How can we write efficient, portable parallel programs for emerging heterogeneous architectures?


Integrated Gen9 GPU graphics + media
CPU core
CPU core
CPU core
CPU core
System Agent (display, memory, I/O controllers)

Pthreads, OpenMP, Vectorized instructions, ...

CUDA, OpenCL, ...

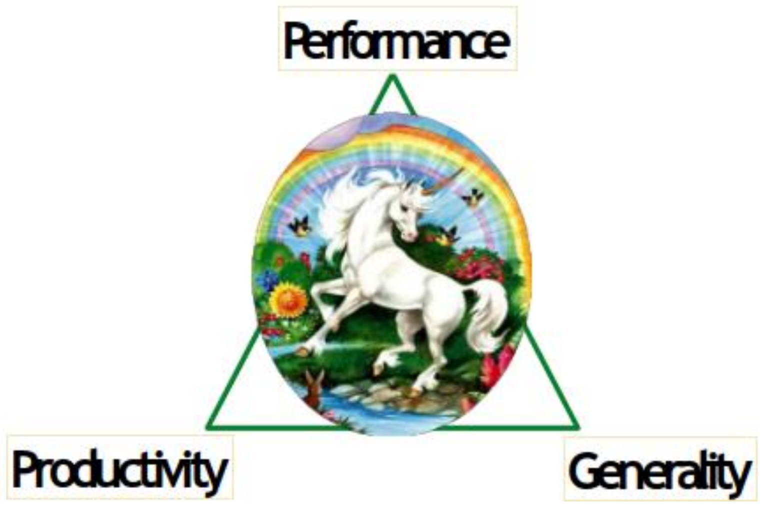


MPI, Spark, ...

Verilog, VHDL, ...

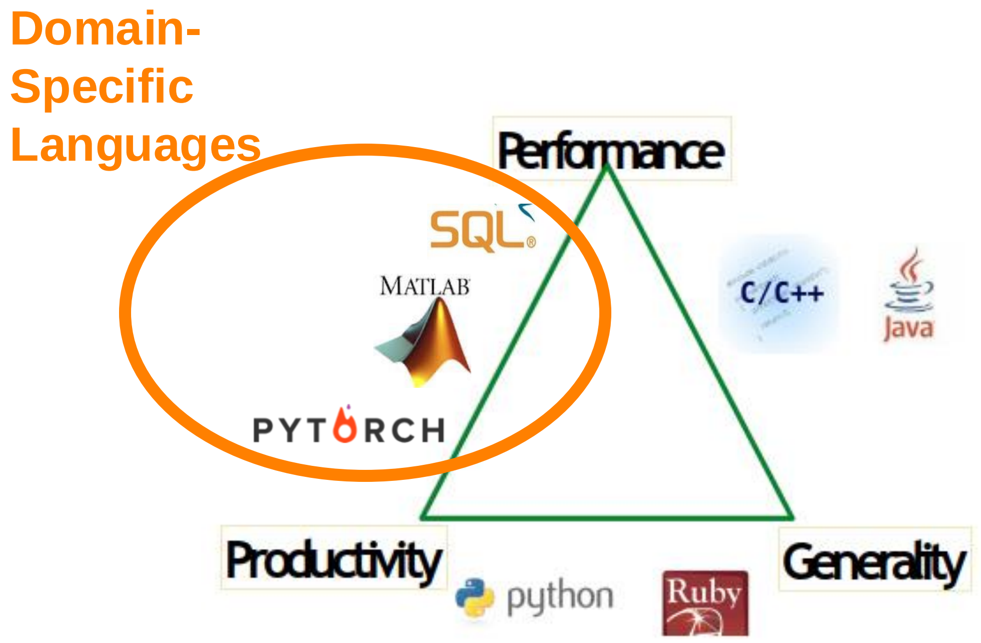
## Heterogeneous parallel programming

- The ideal parallel programming language provides
  - Performance
  - Productivity
  - Generality



## Heterogeneous parallel programming

- Existing languages:



## Domain-Specific Languages

- Programming languages with restricted expressiveness for a particular domain
- High-level, usually declarative, and deterministic
- External or embedded within another language

halide

## Why image processing?

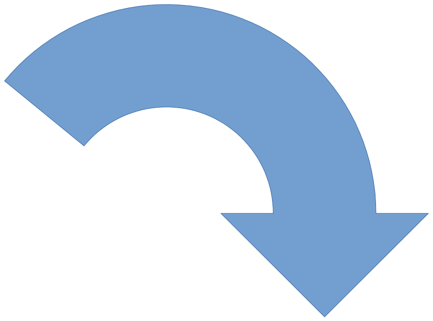
- We are surrounded by data-intensive imaging applications
- Almost every imaging sensor is accessed by a powerful compute unit

Enormous opportunity, demands for **extreme optimization** (performance + **power consumption**)

Keywords: **parallelism, locality**

## Making image processing faster

- Faster algorithms
  - Advanced methods, sometimes approximate computing
- Faster hardware
  - Faster CPUs (higher frequency, higher memory bandwidth, SIMD vector operations)
  - GPUs
- More efficient use of hardware
  - Parallelism
  - Better cache management
  - Memory usage



Require to *reorganize computation*!

## Existing solutions

- C++ with multithreading and SIMD operations
- CUDA/OpenCL for GPUs
- Optimized libraries (BLAS, IPP, MKL, OpenCV)

- Writing efficient image processing pipelines is *hard*!
  - Optimization requires manually transforming program and data structure to exploit parallelism and locality
  - Exploring of different options becomes difficult

## Halide

- Halide proposes a radically different approach:

decouple the **algorithm** from the **schedule**

*what* is computed

fixed once and never modified by scheduling decisions

*where* and *when* it is computed

easily changed to explore different possibilities

## Code example

- C/C++ implementation of a *box blur* algorithm with 3x3 window:

```
void box_filter_3x3(const Image &in, Image &blury){
  Image blurx(in.width(), in.height()); // allocate blurx array

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

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

## Code example

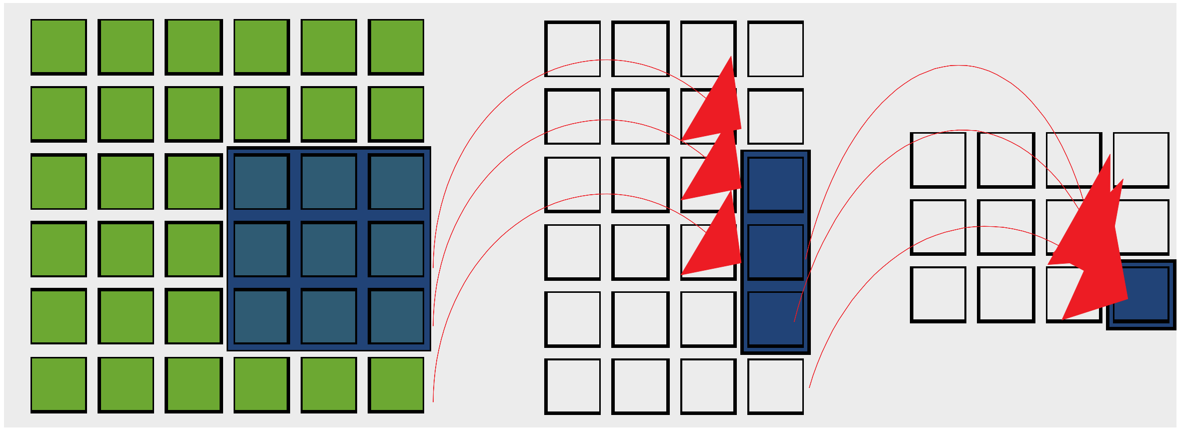
```
void box_filter_3x3(const Image &in, Image &blury){
  Image blurx(in.width(), in.height()); // allocate blurx array

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

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

First average over 3 horizontal neighbors

Then average over 3 vertical neighbors



## Code example

- C/C++ implementation of a *box blur* algorithm with 3x3 window:

```
void box_filter_3x3(const Image &in, Image &blury){
  Image blurx(in.width(), in.height()); // allocate blurx array

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

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

**9.96 ms/megapixel on a quad core x86**

## Code example

❑ Optimized C/C++ implementation:

```
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);
}}}}}
```

**0.9 ms/megapixel on a quad core x86**
11x faster

**parallelism**
(work distributed among threads, SIMD parallel vectors)

## Code example

- Optimized C/C++ implementation:

```
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);
}}}}}
```

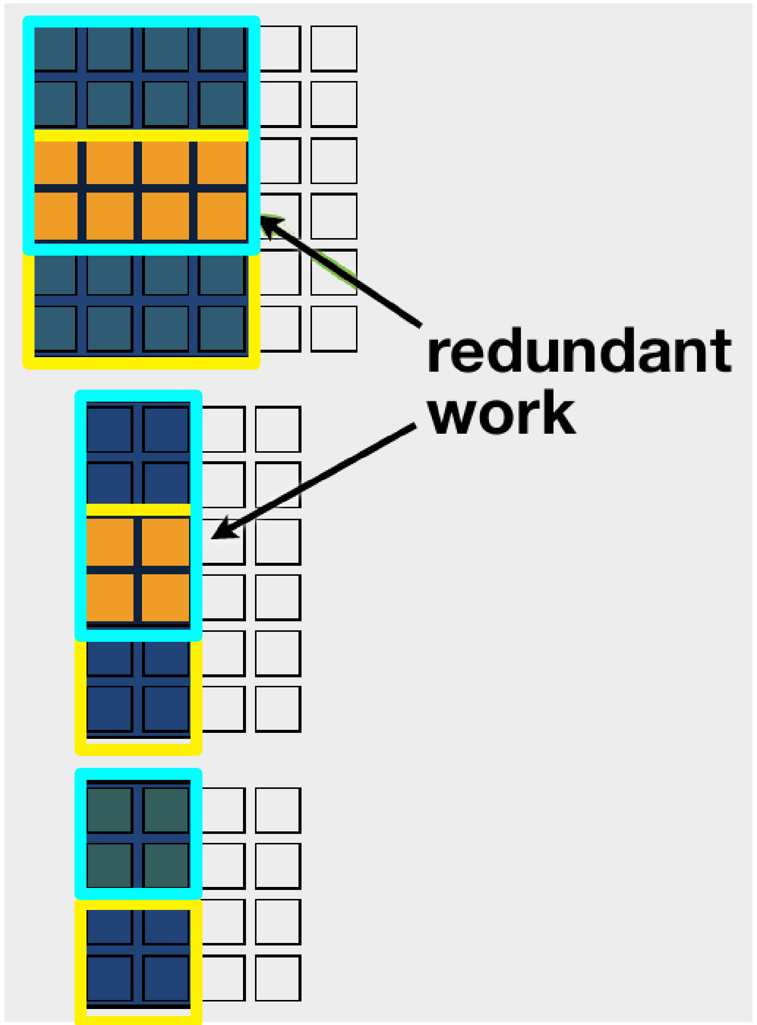
**0.9 ms/megapixel on a quad core x86**
11x faster

**locality**
(compute in tiles,
interleave tiles of blurx, blury,
store blurx in local cache)

## Code example

- So the unoptimized C/C++ version implements the pipeline like this:
  - Compute `blurx` on the whole input image
  - Store the result to memory
  - Load it back and compute `blury` on it
- The optimized version interleaves the execution of the two stages:
  - Compute `blurx` on a chunk of the input
  - Immediately compute `blury` on the intermediate result
  - Throw away the intermediate data, store the result of `blury` and move to the next chunk of the input

## Code example



- Since we immediately throw away the intermediate results, there will be regions of `blurx` that are computed *several times*, once for each neighboring chunk of `blury`
- The same happens between `blurx` and the input image

## Code example

- We were able to move from 9.9 to 0.9ms/megapixel
- The process required knowledge of parallelism and memory hierarchy
- Expensive (time consuming) because it had to be done manually
- Difficult to predict the effect of optimization choices
- Not easily portable to different hardware architectures

Now let’s see how Halide **solves these problems** and at the same time **maintains (or enhances!) the benefits** in terms of performance

## Code example

- Halide **algorithm**:

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

- Halide **schedule**:

```
blury.tile(x, y, xi, yi, 256, 32)
    .vectorize(xi, 8).parallel(y);
blurx.compute_at(blury, x).store_at(blury, x).vectorize(x.8);
```

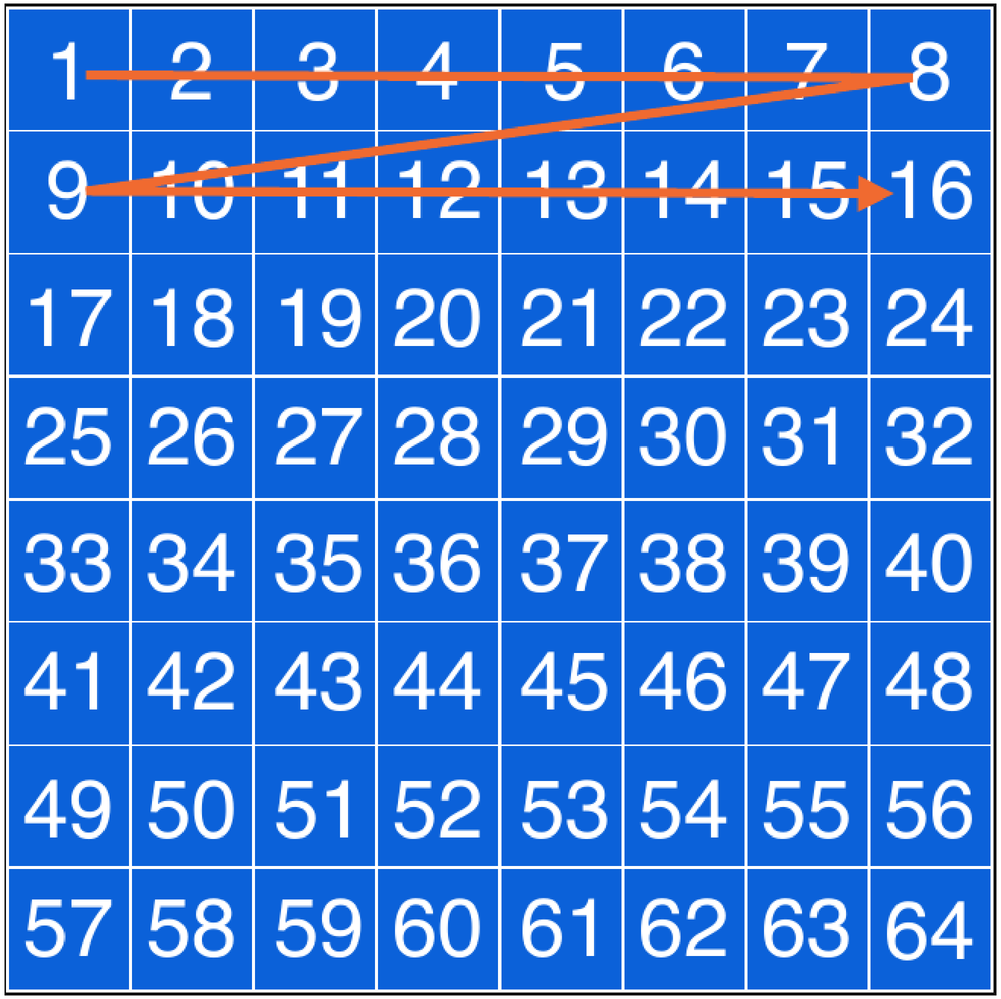
## More on scheduling and performance

- There is a choice space for optimizations in an image processing pipeline
  - In what order should *each stage* compute its values?
  - When should each stage compute its *inputs*?

## More on scheduling and performance

- In what order should I compute my values?

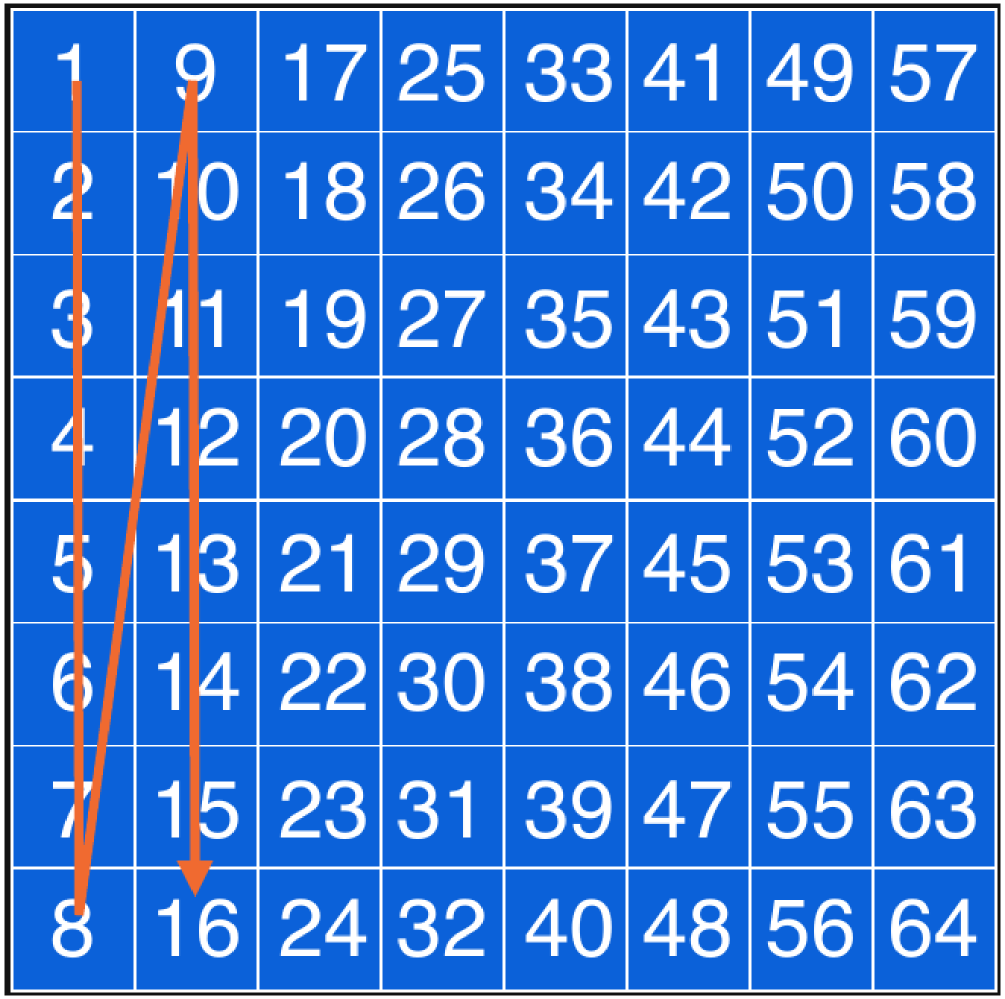
Serial y,
serial x



## More on scheduling and performance

- In what order should I compute my values?

Serial x,
serial y

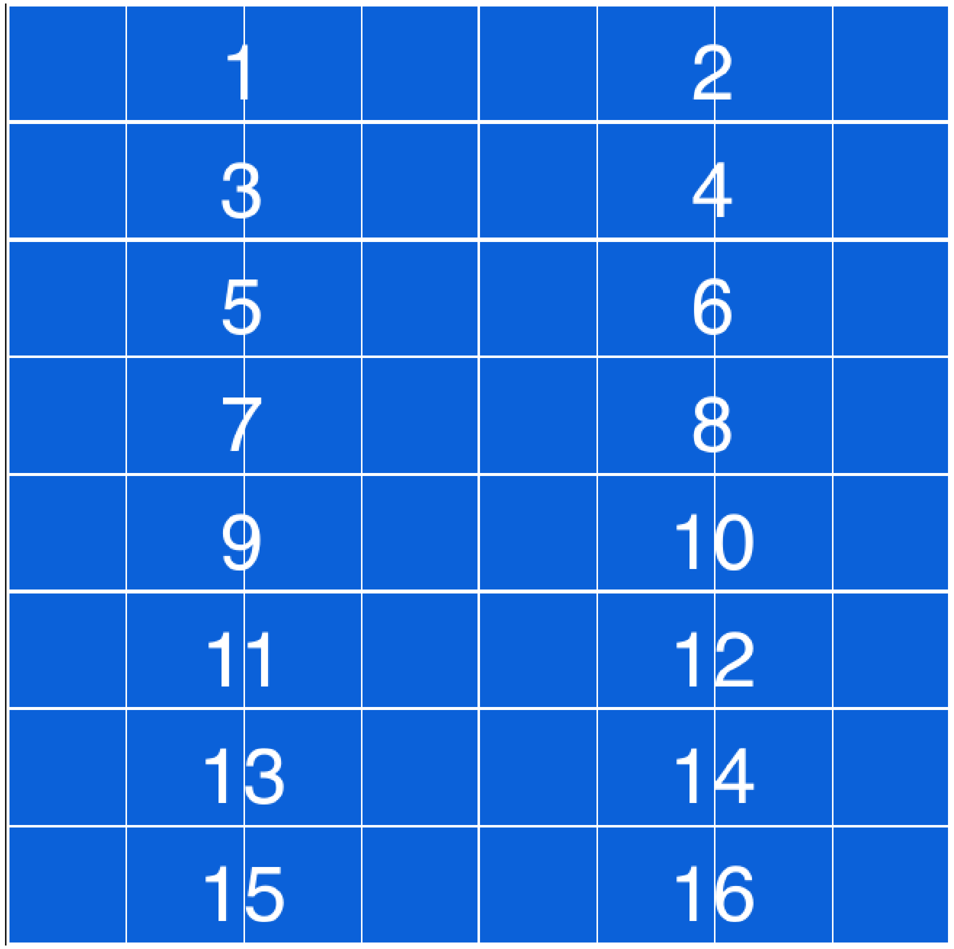


| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 9 | 17 | 25 | 33 | 41 | 49 | 57 |
| 2 | 10 | 18 | 26 | 34 | 42 | 50 | 58 |
| 3 | 11 | 19 | 27 | 35 | 43 | 51 | 59 |
| 4 | 12 | 20 | 28 | 36 | 44 | 52 | 60 |
| 5 | 13 | 21 | 29 | 37 | 45 | 53 | 61 |
| 6 | 14 | 22 | 30 | 38 | 46 | 54 | 62 |
| 7 | 15 | 23 | 31 | 39 | 47 | 55 | 63 |
| 8 | 16 | 24 | 32 | 40 | 48 | 56 | 64 |

## More on scheduling and performance

- In what order should I compute my values?

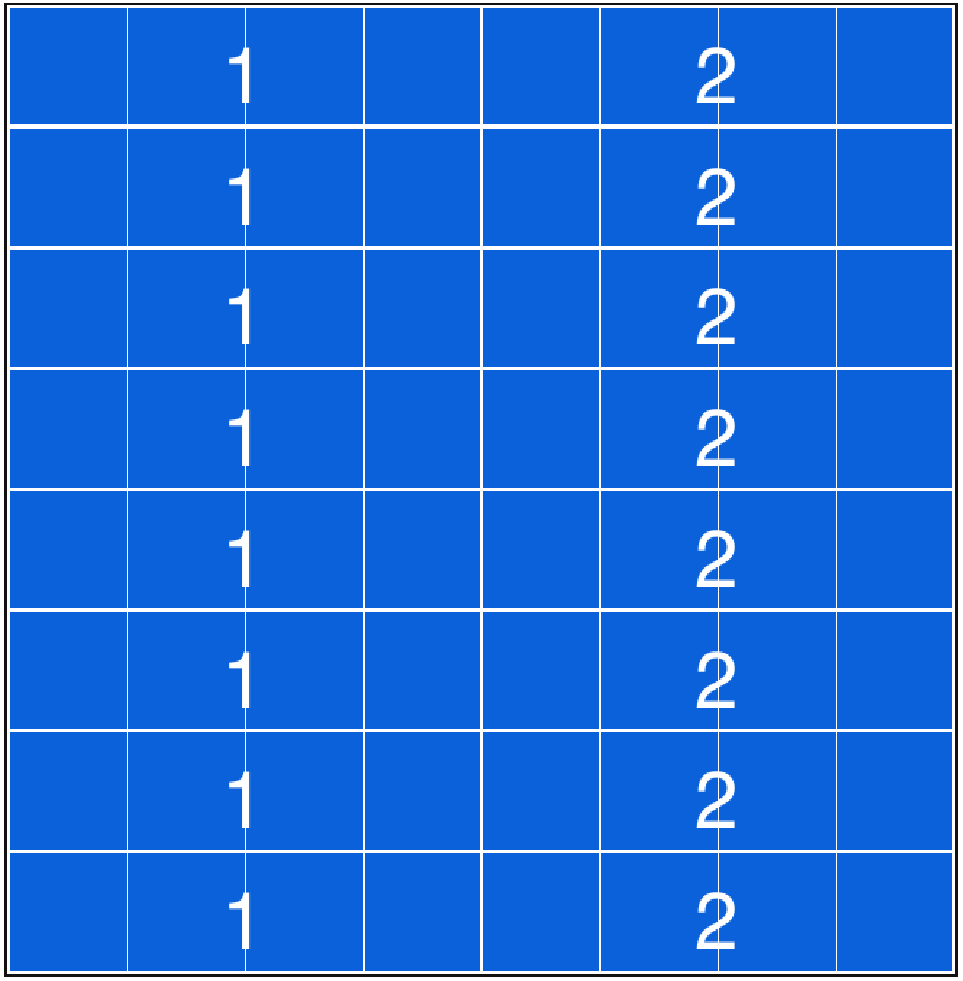
Serial y,
Vectorize x by 4



## More on scheduling and performance

- In what order should I compute my values?

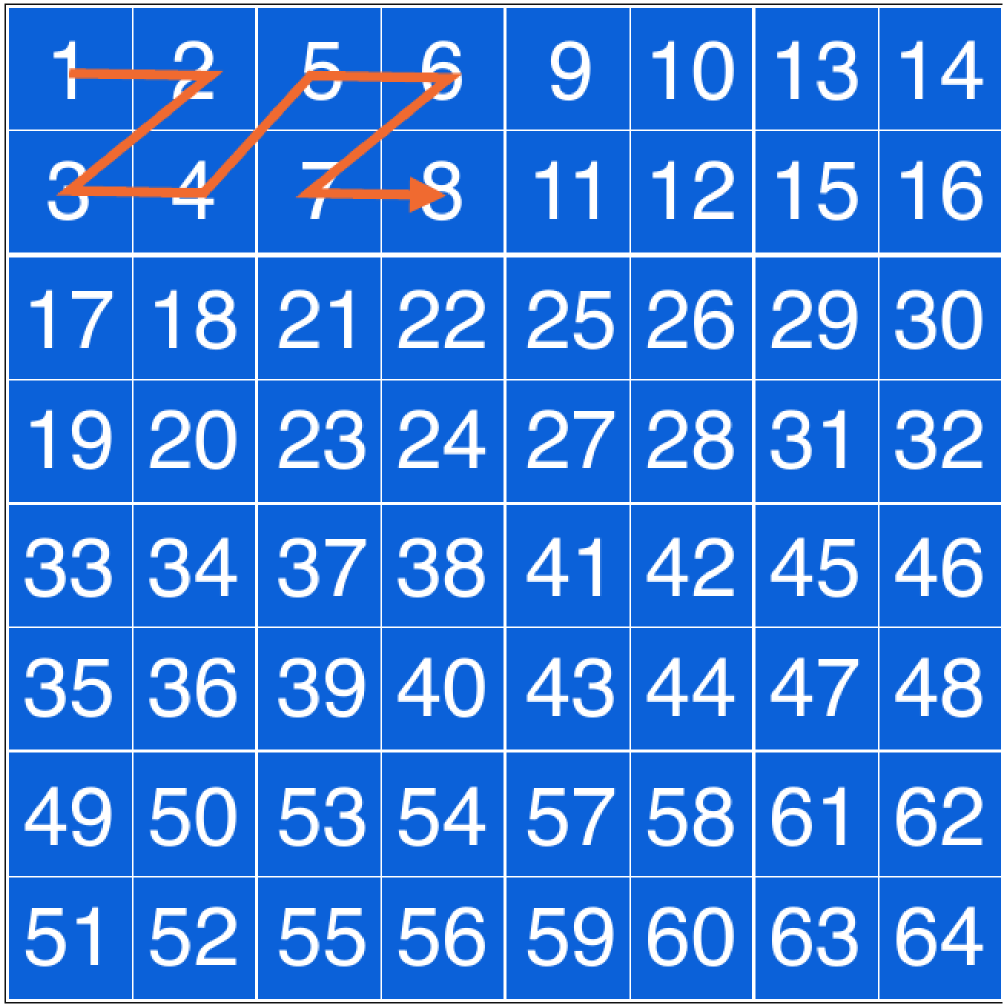
Parallel y,
Vectorize x by 4



# More on scheduling and performance

- In what order should I compute my values?

Split x by 2,
Split y by 2,
Serial $y_{outer}$,
Serial $x_{outer}$,
Serial $y_{inner}$,
Serial $x_{inner}$



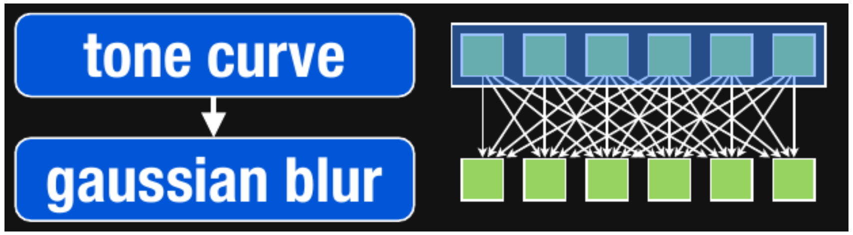
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 5 | 6 | 9 | 10 | 13 | 14 |
| 3 | 4 | 7 | 8 | 11 | 12 | 15 | 16 |
| 17 | 18 | 21 | 22 | 25 | 26 | 29 | 30 |
| 19 | 20 | 23 | 24 | 27 | 28 | 31 | 32 |
| 33 | 34 | 37 | 38 | 41 | 42 | 45 | 46 |
| 35 | 36 | 39 | 40 | 43 | 44 | 47 | 48 |
| 49 | 50 | 53 | 54 | 57 | 58 | 61 | 62 |
| 51 | 52 | 55 | 56 | 59 | 60 | 63 | 64 |

## More on scheduling and performance

- When should I compute my inputs?

```
compute blury:
  for ...:
    blury(...) = ...
```

```
compute blurx:
  for ...:
    blurx(...) = ...
```



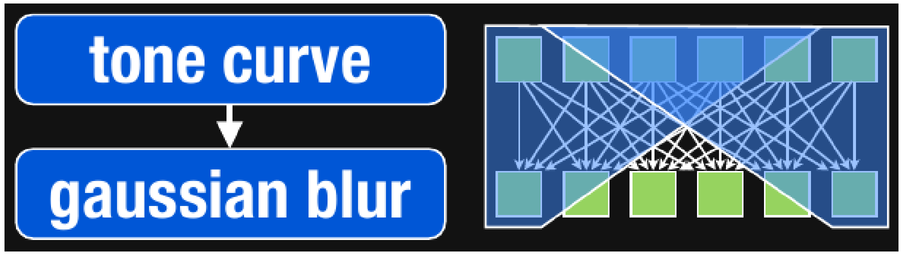
Compute and store producer, then compute consumer

+ no recomputation
- poor locality

## More on scheduling and performance

- When should I compute my inputs?

```
compute blurx:
  for c:
    for y:
      for x:
        blurx(...) = ... ... ...
                     ... ... ...
                     ... ... ...
                     ... ... ...
                     ... ... ...
                     ... ... ...
                     ... ... ...
```

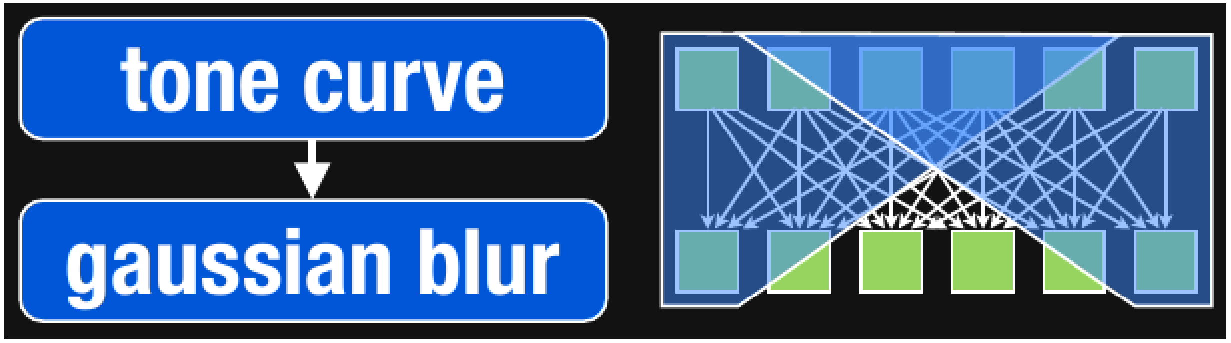


Compute producer values when needed by consumer, then throw them away

+ locality
- redundant work

## More on scheduling and performance

- When should I compute my inputs?

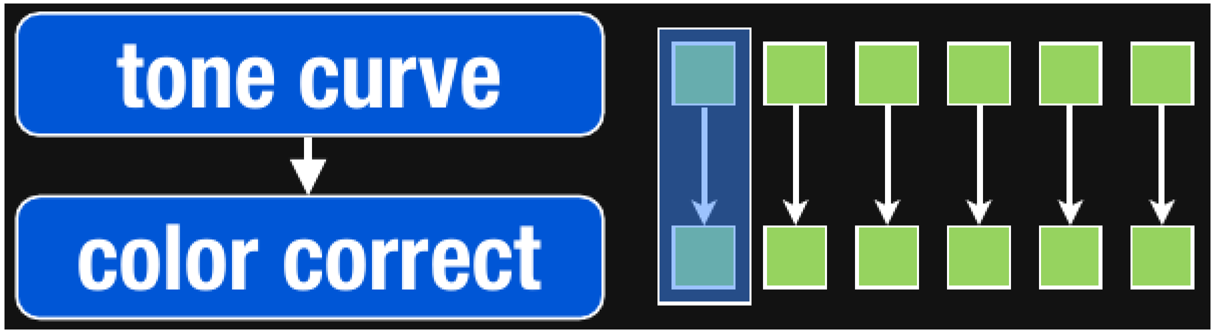


Compute producer values when needed by consumer, then keep the old values for reuse?

+ locality
+ no redundant work
- poor parallelism

## More on scheduling and performance

- When should I compute my inputs?



Compute producer values when needed by consumer, then throw them away

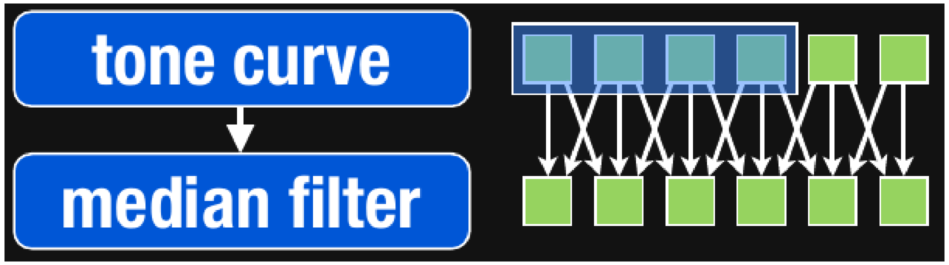
+ locality

+ no redundant work (property of the consumer function)

## More on scheduling and performance

- When should I compute my inputs?

```
compute blurx:
  for c:
    for y:
      compute blury:
        for x:
          blury(...) = ...
      for x:
        blurx(...) = ...
```

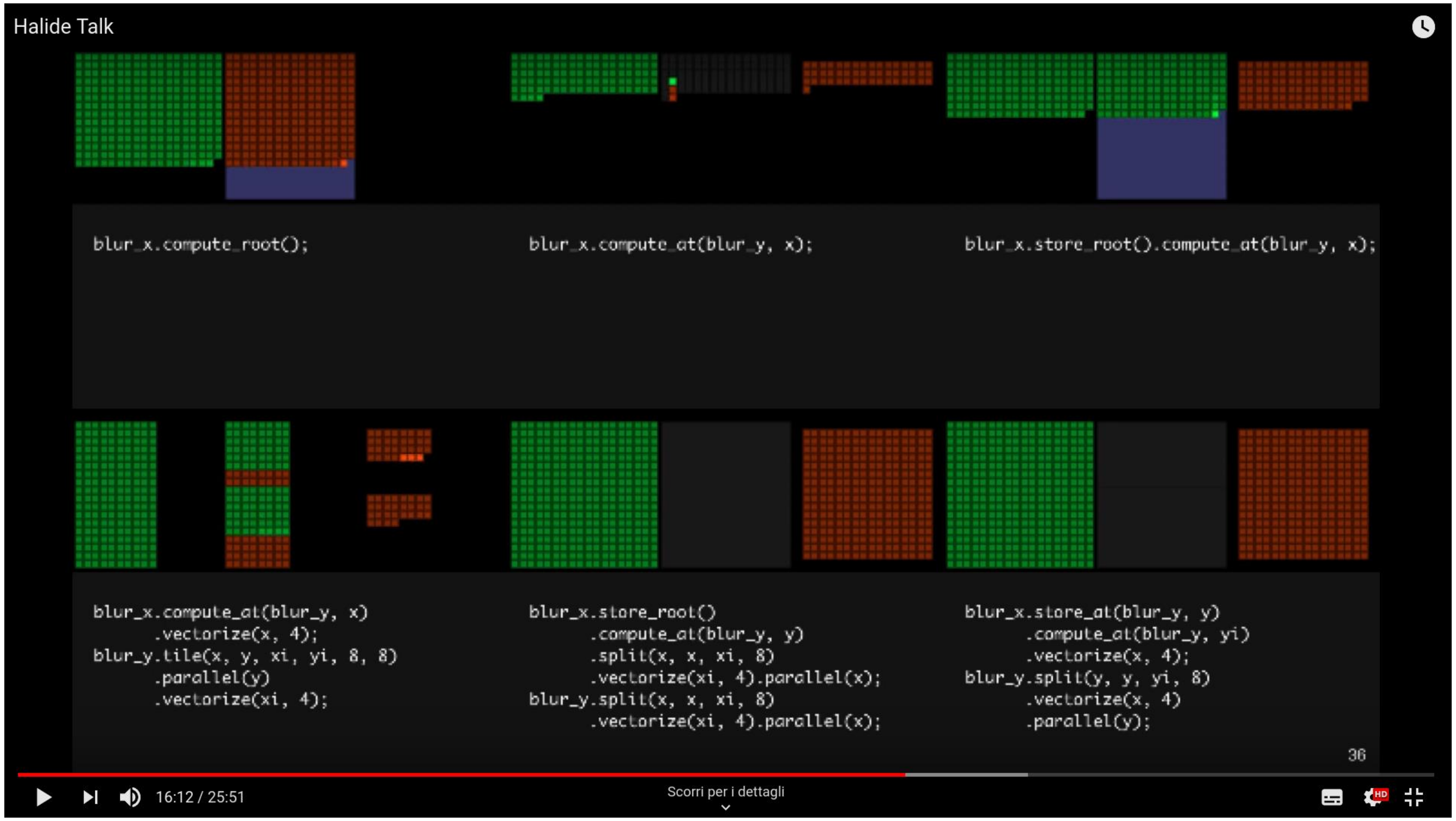


Compute a subset of producer values all at once, then pass them to the consumer

+/- depend on chunk size, kernel size, parallelism...

## More on scheduling and performance

Animated visualization of these and other scheduling options:
https://www.youtube.com/watch?time_continue=14&v=3uiEyEKji0M&feature=emb_logo



```
blur_x.compute_root();
```

```
blur_x.compute_at(blur_y, x);
```

```
blur_x.store_root().compute_at(blur_y, x);
```

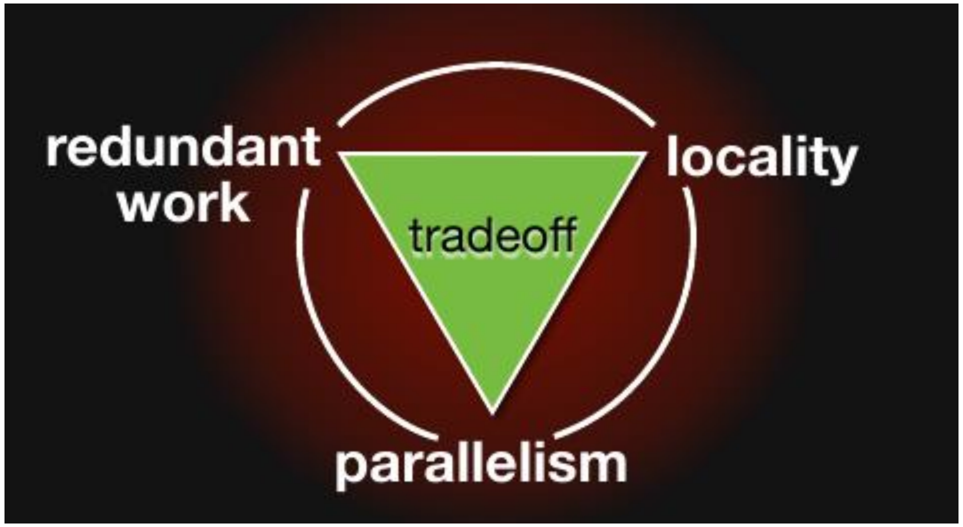
```
blur_x.compute_at(blur_y, x)
      .vectorize(x, 4);
blur_y.tile(x, y, xi, yi, 8, 8)
      .parallel(y)
      .vectorize(xi, 4);
```

```
blur_x.store_root()
      .compute_at(blur_y, y)
      .split(x, x, xi, 8)
      .vectorize(xi, 4).parallel(x);
blur_y.split(x, x, xi, 8)
      .vectorize(xi, 4).parallel(x);
```

```
blur_x.store_at(blur_y, y)
      .compute_at(blur_y, yi)
      .vectorize(x, 4);
blur_y.split(y, y, yi, 8)
      .vectorize(x, 4)
      .parallel(y);
```

## More on scheduling and performance

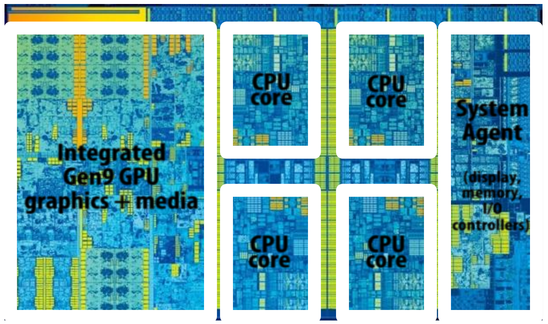
- Performance requires **complex trade-offs** and frequent **reorganization of computation**



- Halide makes this iterative process easier

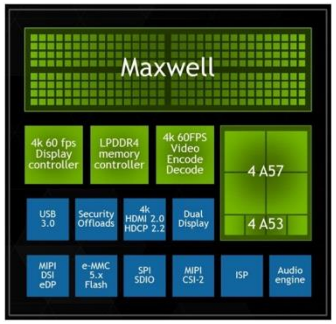
## Heterogeneous parallel programming

- How can we write efficient, portable parallel programs for emerging heterogeneous architectures?


Integrated Gen9 GPU graphics + media
CPU core
CPU core
CPU core
CPU core
System Agent
(display, memory, I/O controllers)

Pthreads, OpenMP, Vectorized instructions, ...

CUDA, OpenCL, ...

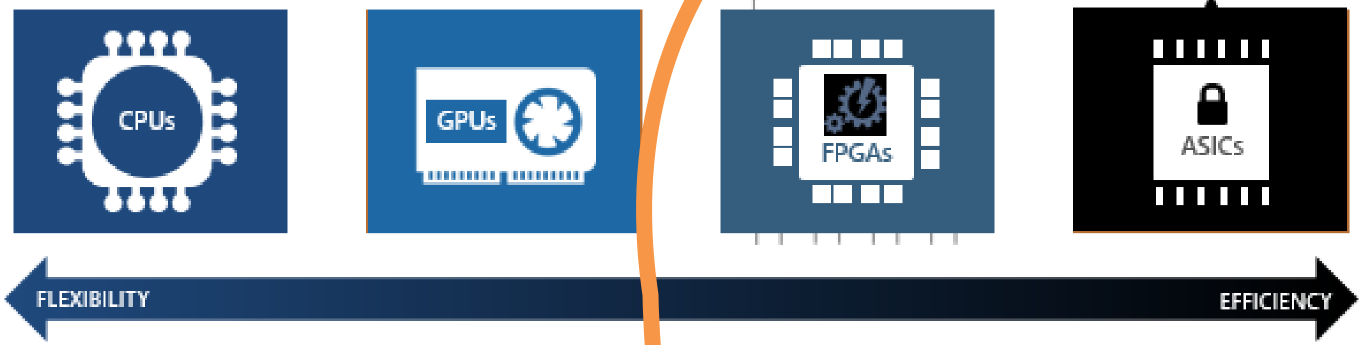


MPI, Spark, ...

Verilog, VHDL, ...

## Heterogeneous processing

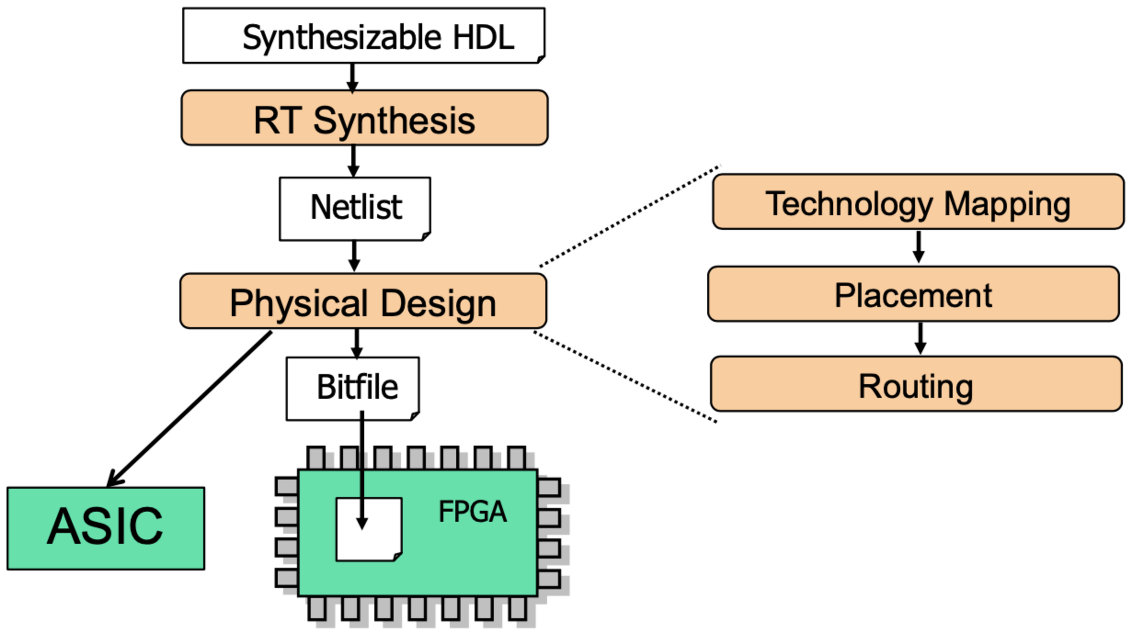
- Choosing the right tool for the job:



| Control-oriented | Throughput-oriented | Reconfigurable | Fixed function |
|---|---|---|---|
| | x10 more efficient | x100 more efficient? | x100-1000 more efficient |
| Easiest to program | | Difficult to program (active research on how to make it easier) | Not programmable<br>Costs 10-100M $ to design/verify/ manufacture |

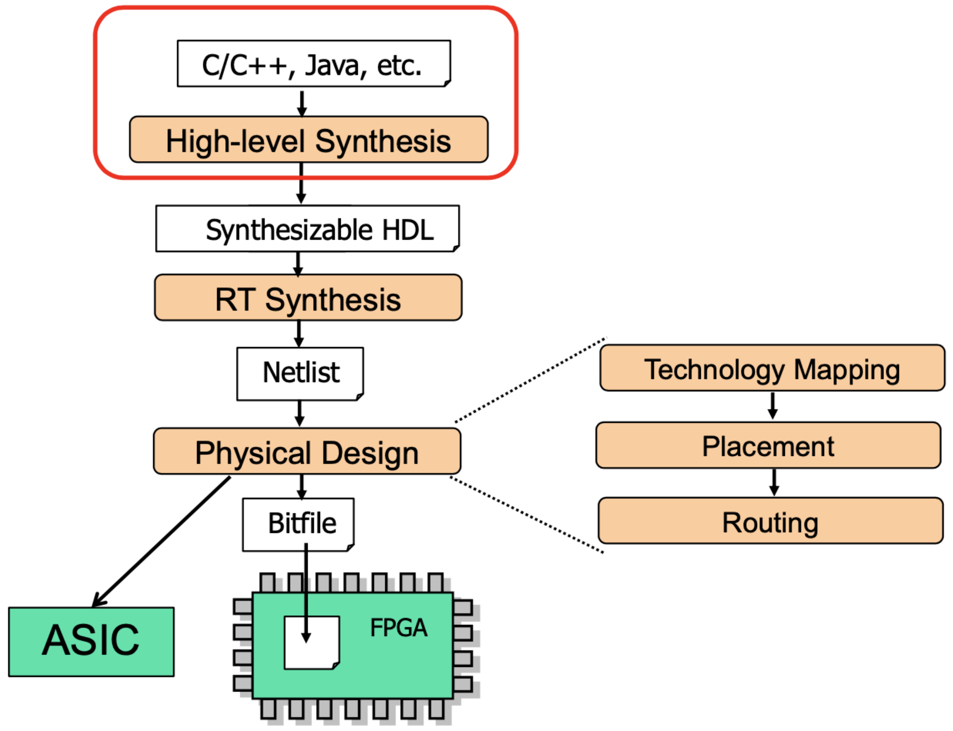
## “Traditional” FPGA/ASIC design flow

- Register-Transfer Level (RTL) design
  - Specify behavior and structure (muxes, registers…) in Verilog/VHDL
  - Precise, fine-grained control
  - Time-consuming and error-prone!



# High-Level Synthesis

- High-level behavioral design through sw programming languages
  - Automated translation to RTL
  - Like compilers, but for hardware



## Benefits

- Number of expert SW developers vs HW designers
- Easier to describe complex functionalities
- Technology-independent design
- Easier HW/SW partitioning
- Faster design cycle (including verification)

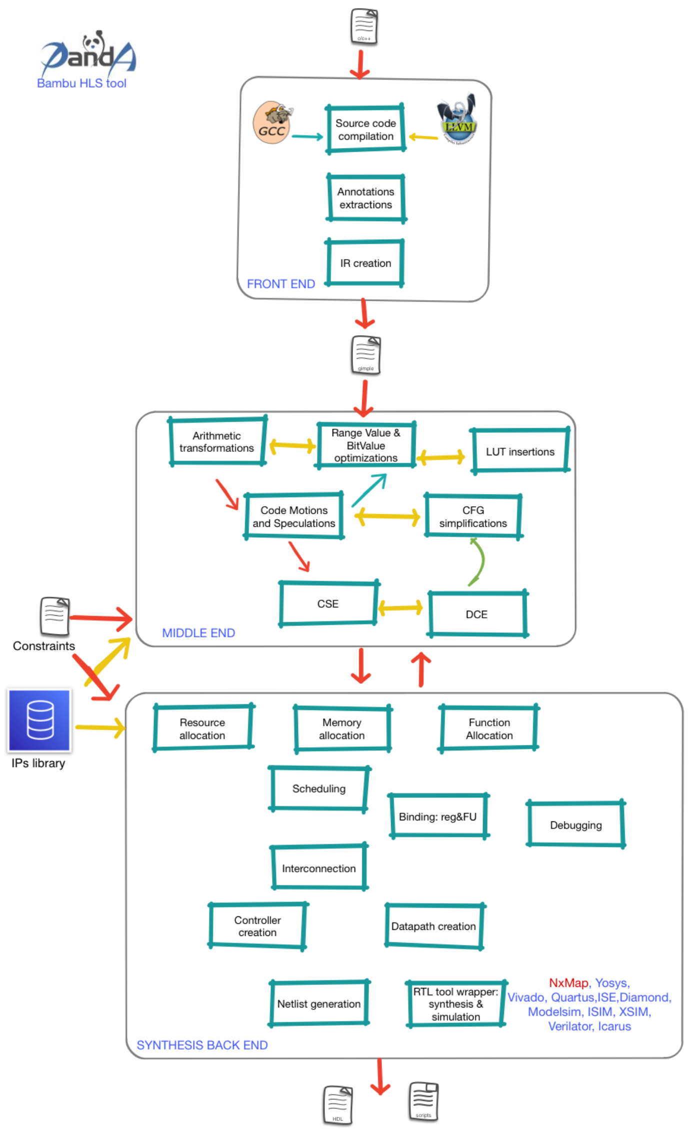
## Challenges

- While compilers map instructions to existing hardware, HLS tools must create a custom architecture
- Huge design space to explore
- Not all code is suitable to be translated into hardware

## High-Level Synthesis

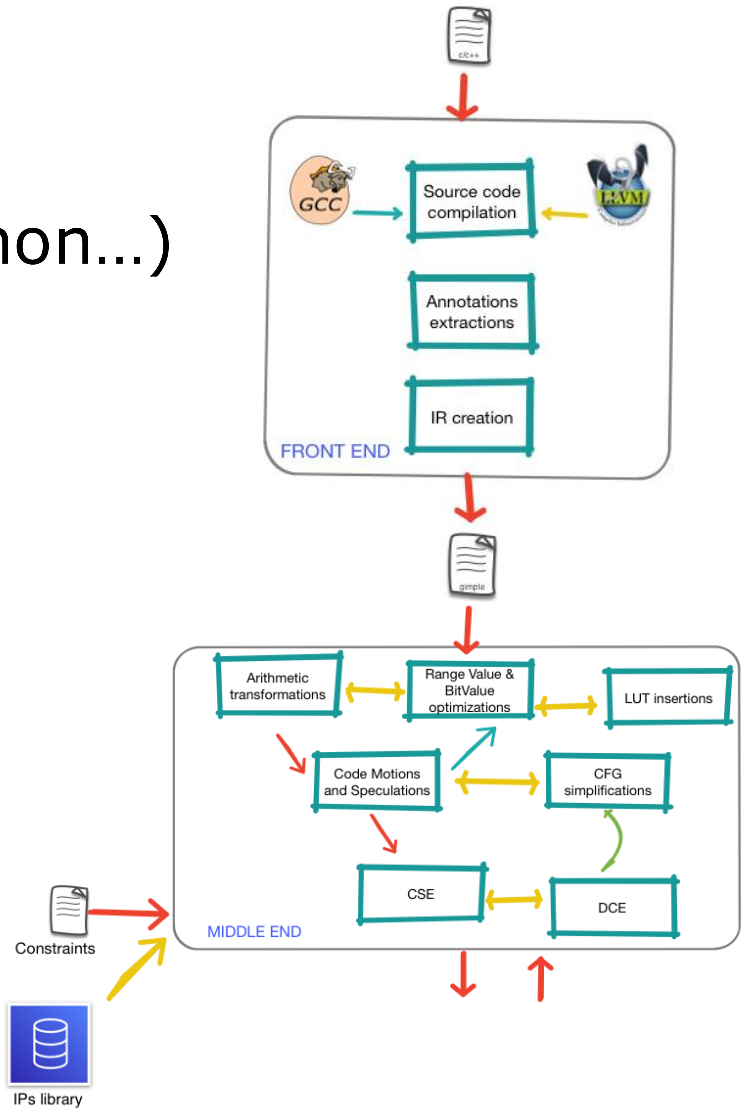
- Example HLS tool: Bambu, developed at DEIB
- (Probably) Most popular: Vitis HLS, from AMD/Xilinx
- Others: Intel HLS Compiler, SmartHLS (formerly LegUp) …

- Compiler-like structure: front-end, middle-end, back-end
- Transformations applied on an Intermediate Representation (IR) of the input



# High-Level Synthesis

❑ HLS inputs:
- High-level language specification (C/C++/Python…)
- Library of *characterized* modules
- Constraints (area/delay)
- (Optional) optimization directives



## High-Level Synthesis

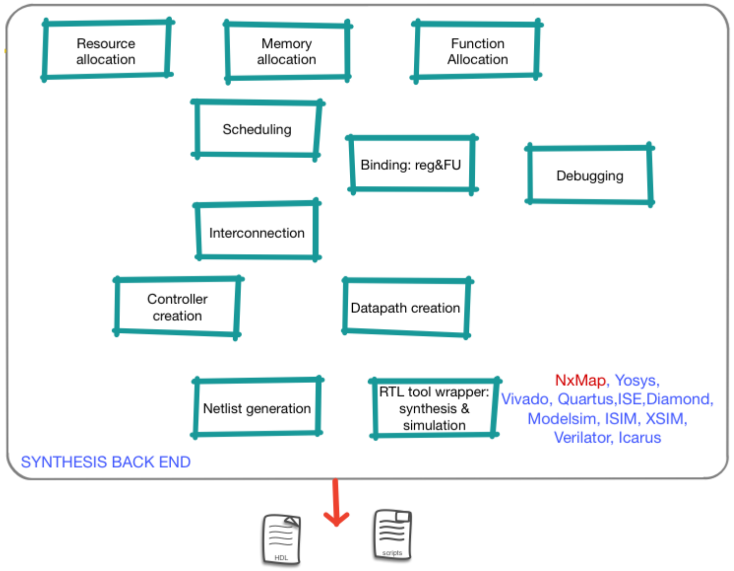
- HLS objectives:
  - Minimum area (functional units, registers, memory, interconnect)
  - Maximum speed (latency in clock cycles, clock period, throughput)

Generally, one parameter is set as a constraint, and the other one is optimized

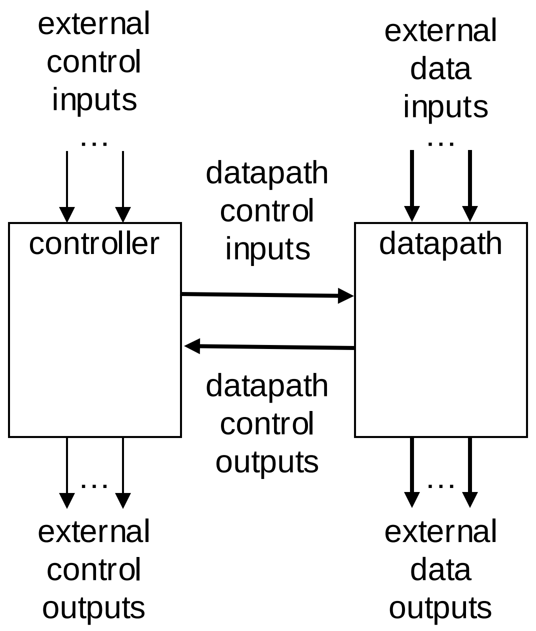
- Additional goals:
  - Low power consumption, testability, more accurate estimation, fault tolerance…

## High-Level Synthesis

- HLS output:
  - Synthesizable Verilog/VHDL
  - Controller + datapath (FSMD)
  - Next steps: simulation, logic synthesis, implementation



# The FSMD model



controller and datapath

**Data-path:**

- Functional resources: Perform operations on data (arithmetic and logic blocks).
- Memory resources: Store data (memory and registers).
- Interconnection resources: (muxes, busses and ports).

**Controller:**

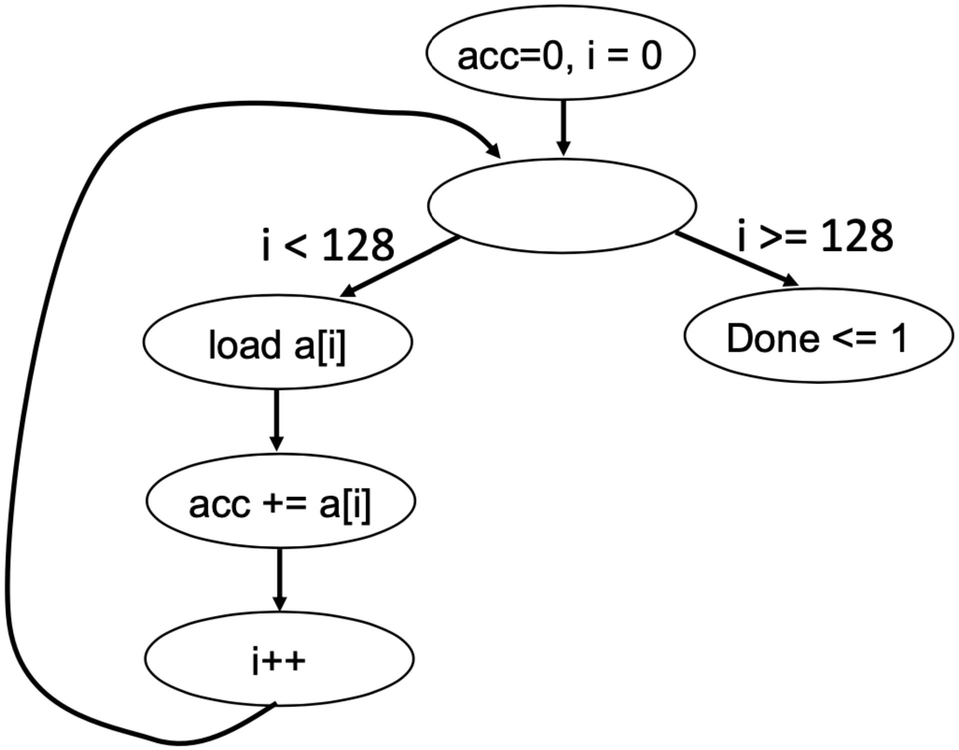
- Finite state machine (FSM)
- Controller + microprogram
- Synchronization scheme (e.g., global clock single phase with master-slave registers)

## Manual example

- From high-level code to circuits:
  1. Build FSM for the controller
  2. Build datapath

```
acc = 0;
for (i=0; i < 128; i++)
   acc += a[i];
```

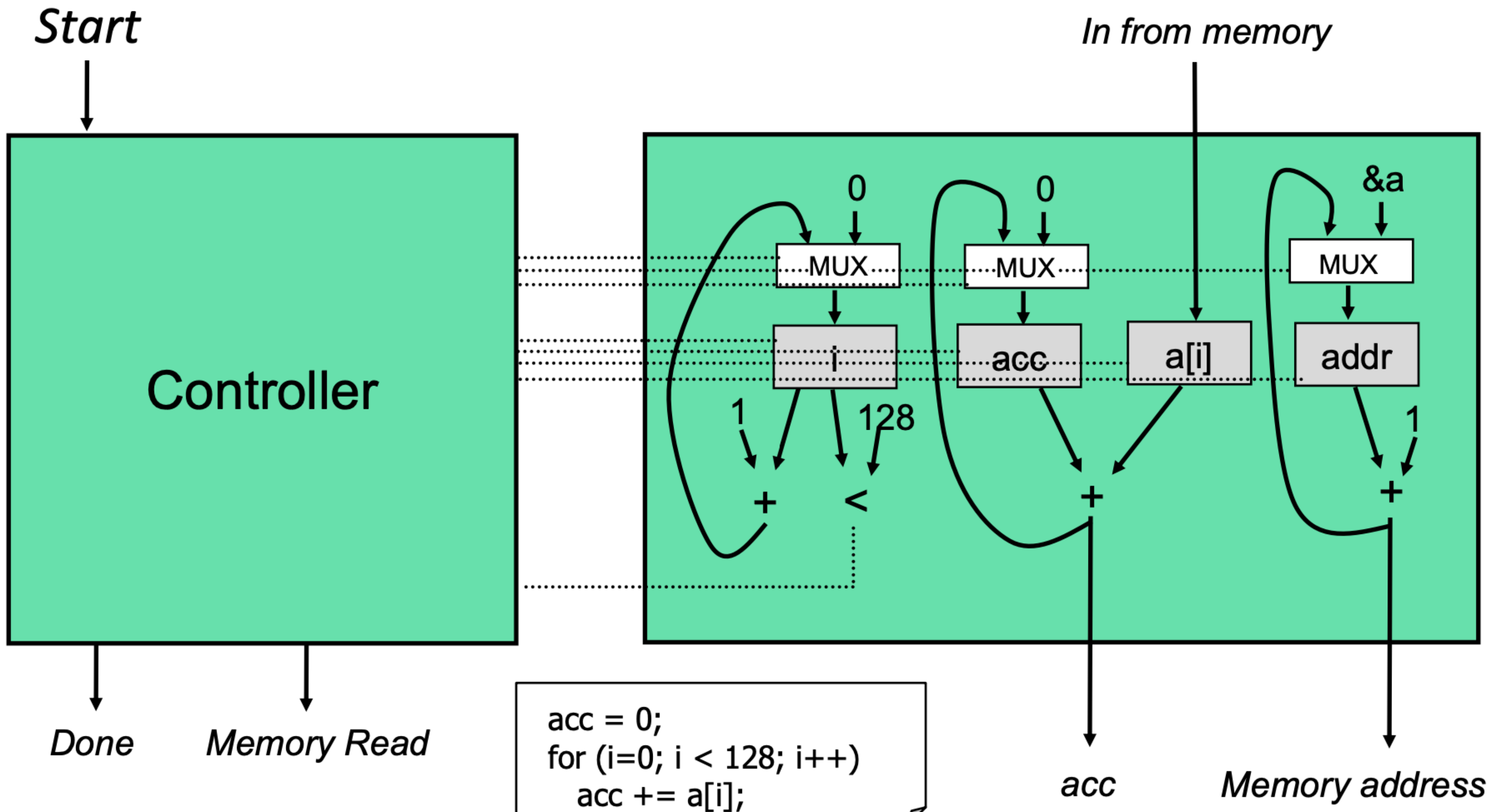
Input code



FSM diagram

## Manual example

- Splitting controller and datapath

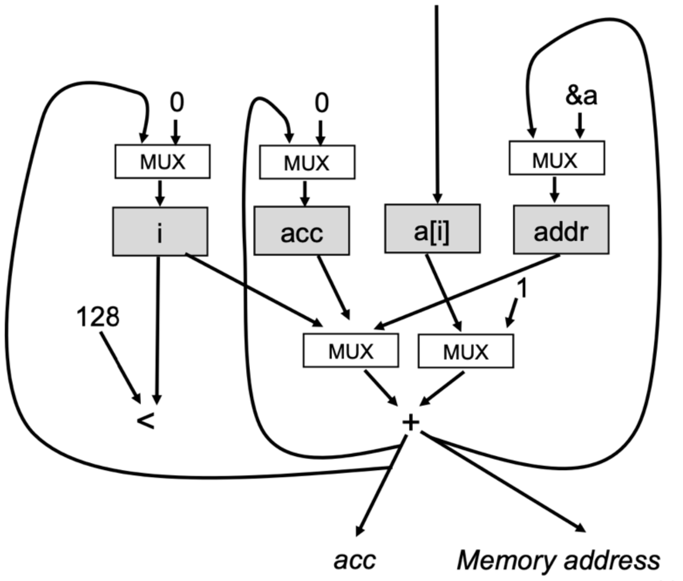


```
acc = 0;
for (i=0; i < 128; i++)
   acc += a[i];
```

## Manual example

- HLS performs these steps automatically
  - Scheduling (determine when to perform each operation)
  - Resource allocation (allocate resources for each operation)
  - Binding (map operations to resources)

- Possible optimization: one less adder
- More optimization opportunities in the input program, in the FSMD, in the datapath…



## HLS IR basic concepts

- **Single Static Assignment form (SSA)**
  - A program is in SSA form if every variable is only assigned once
- **3-address form**
  - Single operator and at most three names

**Original**

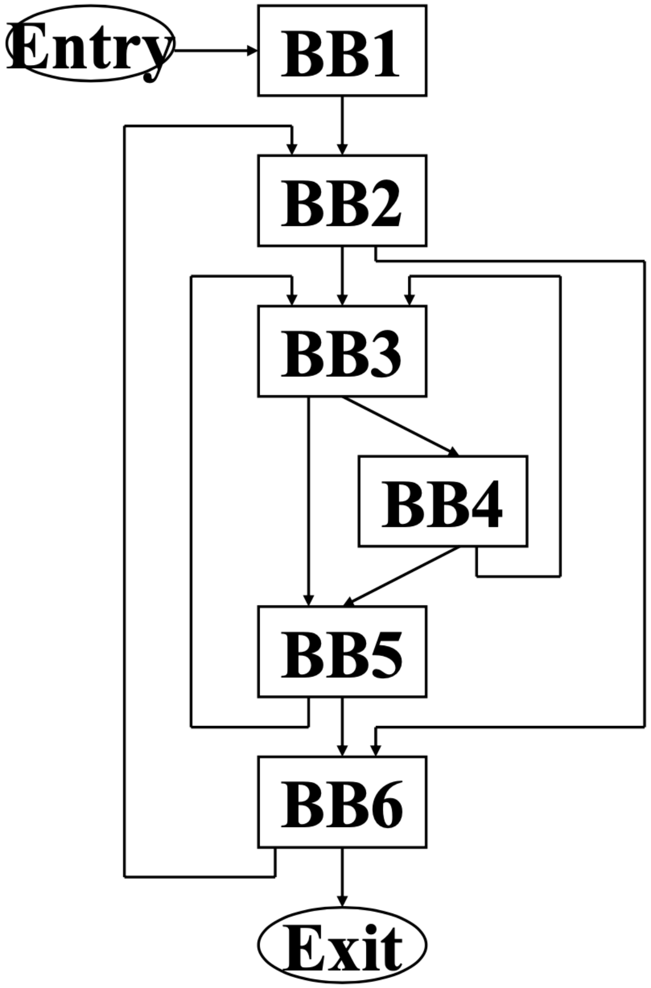
```
a  := b + c
b  := c + 1
d  := b + c
a  := a + 1
e  := a + b
```

**SSA**

$a_1 := b_1 + c_1$
$b_2 := c_1 + 1$
$d_1 := b_2 + c_1$
$a_2 := a_1 + 1$
$e_1 := a_2 + b_2$

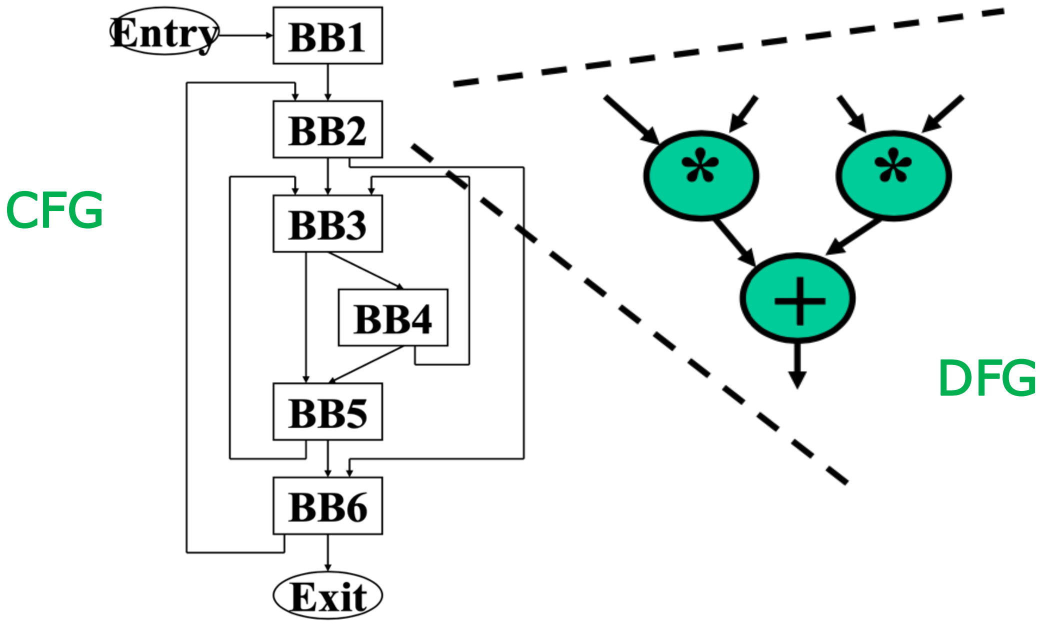
## HLS IR basic concepts

- Basic block
  - Sequence of instructions with no labels (except at the first instruction) and no jumps (except at the last instruction)
- Control Flow Graph
  - Nodes: basic blocks
  - Edges: potential flow of control (loop, if/else, goto…)



## HLS IR basic concepts

- Data Flow Graph
  - Nodes: operations
  - Edges: data dependencies (read-after-write)
  - One DFG for each basic block



## HLS scheduling

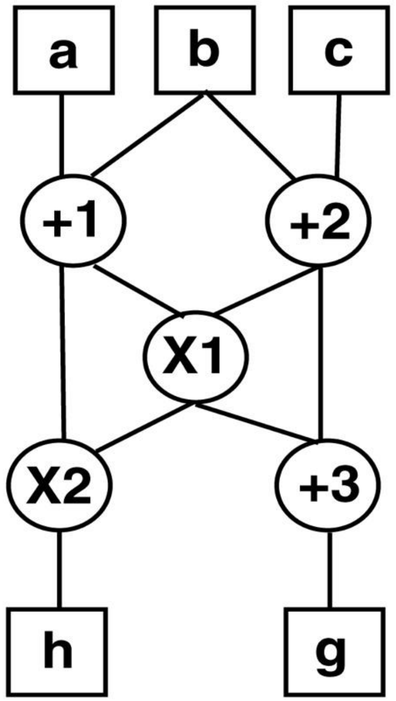
- Definition: assignment of operations to time (control steps)
- Possibly within limits on hardware resources and timing
- Exploits potential parallelism
- Scheduling problems are *NP-hard*, so many techniques and heuristics can be used:
    - ASAP – As soon as possible
    - ALAP – As late as possible
    - List-based scheduling (resource constrained)
    - Integer Linear Programming
    - …

## HLS scheduling

- Inputs:
  - IR (CDFG, SSA form)
  - Operation delays in cycles
  - Timing/resource constraints
- Output:
  - Start time for every operation in the IR
- Goal:
  - Best latency/area trade-off

## HLS scheduling

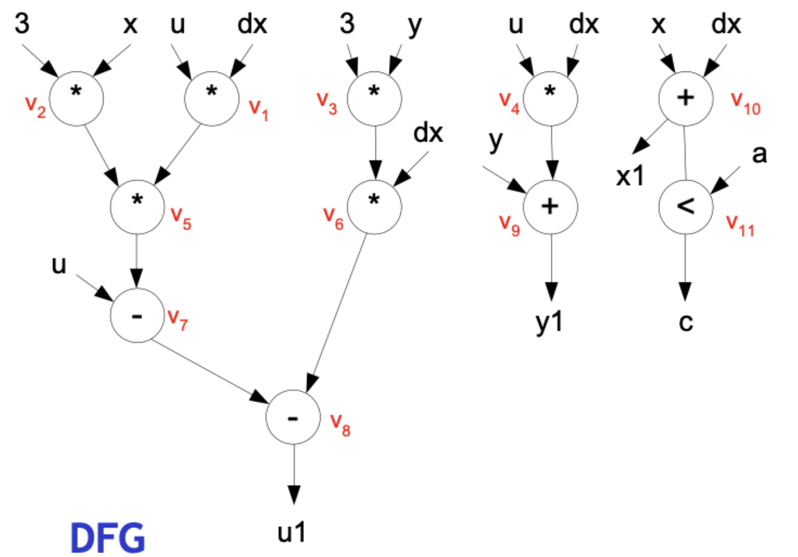
- Example of different scheduling solutions



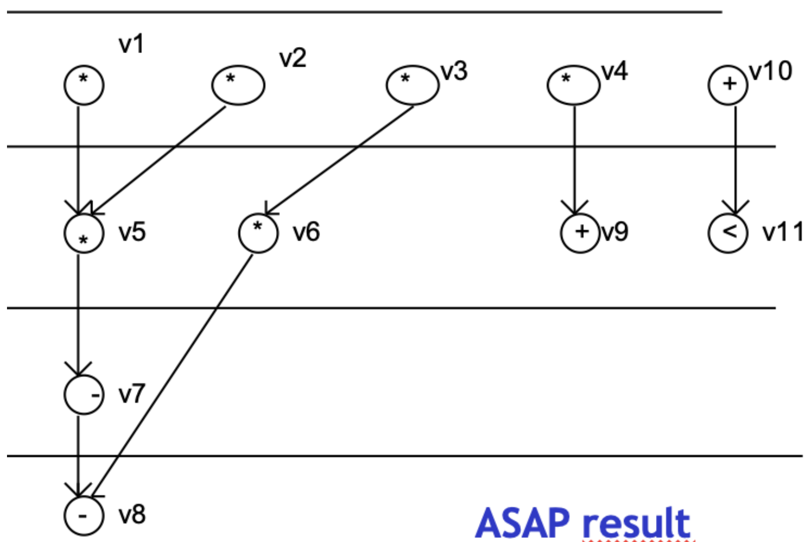
| 1 adder & 1 mult. | 2 adder & 1 mult. |
|---|---|
| step 1: +1 | step 1: +1, +2 |
| step 2: +2 | step 2: X1 |
| step 3: X1 | step 3: +3, X2 |
| step 4: +3, X2 | |

## HLS scheduling

- Simplest model: Minimum Latency Unconstrained Scheduling
  - All operations have bounded delays expressed as number of clock cycles, no area constraints
- ASAP scheduling



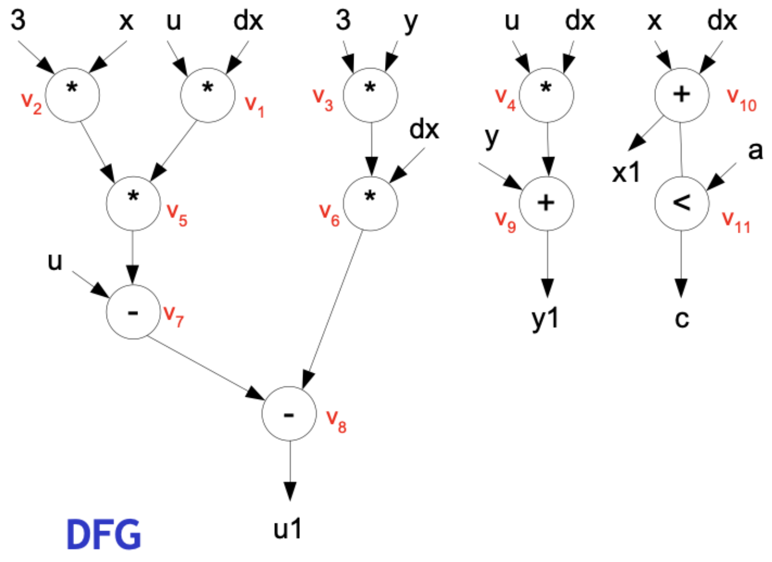
DFG



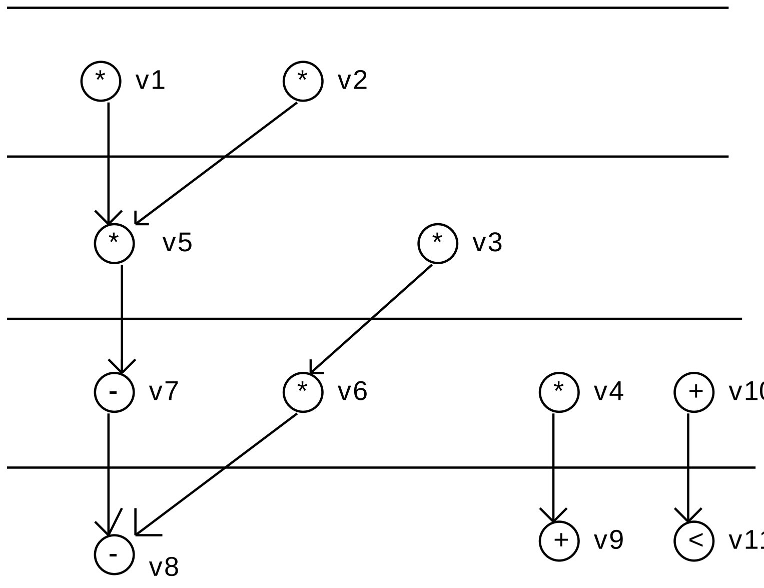
ASAP result

## HLS scheduling

- Dual algorithm - ALAP scheduling
  - Latency-constrained scheduling
  - Maximum latency = ASAP latency



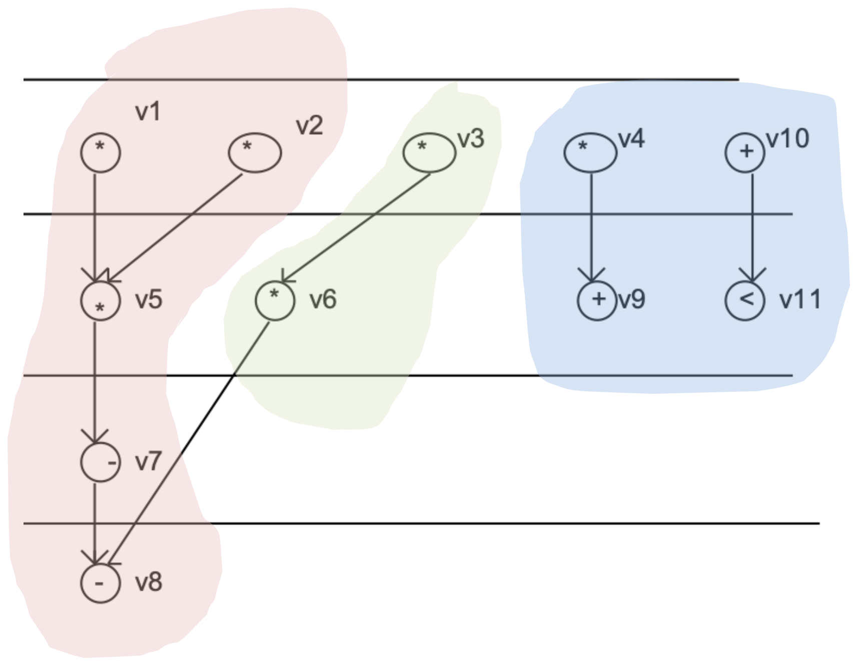
DFG

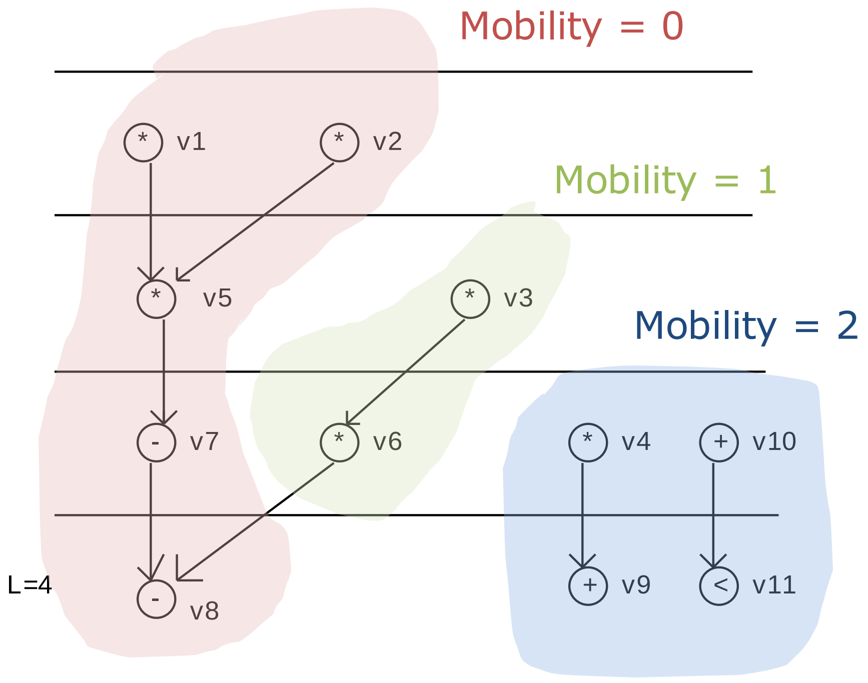


ALAP result

## HLS scheduling

- **Mobility:**
  - Difference between ASAP and ALAP start time
  - Operations with mobility = 0 are on the critical path





## HLS scheduling

- More realistic model: Constrained Scheduling
  - All operations have bounded delays expressed as number of clock cycles
  - Minimize latency given constraints on resources (ML-RCS)
  - Minimize resources given a bound on latency (MR-LCS)
- NP-hard
  - Exact methods (ILP)
  - Heuristics (list-based, force-directed)

## HLS scheduling

- List-based Scheduling
  - Heuristic method for both ML-RCS and MR-LCS
  - Does not guarantee optimal solution
  - Greedy strategy, O(n) time complexity
- Algorithm:
  1. Construct a priority list (based e.g. on operation mobility)
  2. While not all operations scheduled:
     1. For each available resource, select an operation in the ready list following the descending priority
     2. Assign these operations to the current clock cycle
     3. Update the ready list
     4. Clock cycle ++

## HLS scheduling

- Exercise: schedule the given basic block from an HLS IR using list-based scheduling.

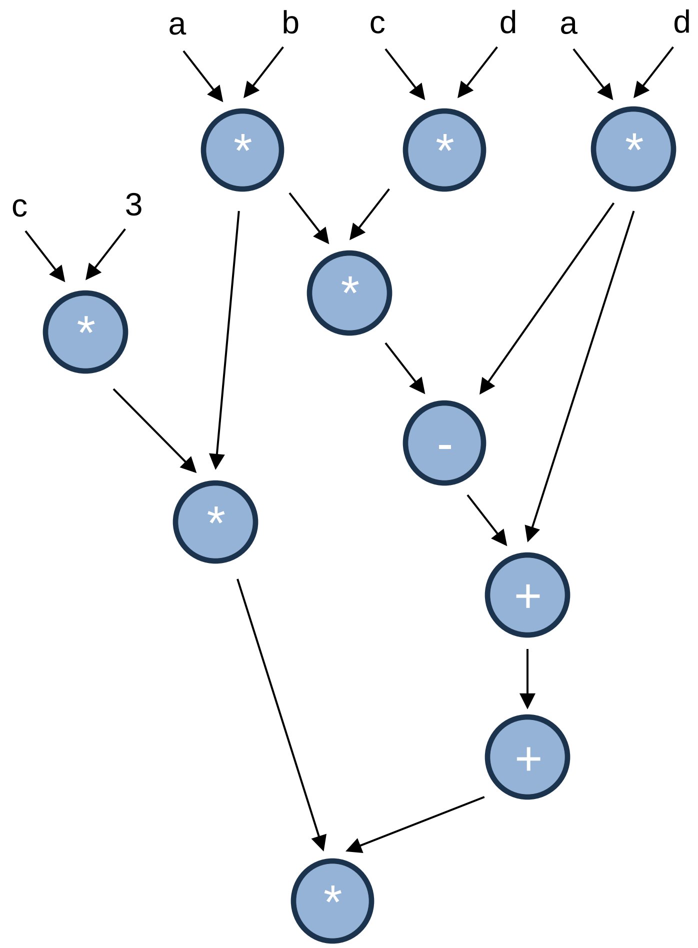
```
tmp_1 = a * b;
tmp_2 = c * d;
x_1 = tmp_1 * tmp_2;
x_2 = a * d;
x_3 = x_2 - x_1;
x_4 = x_2 + x_3;
x_5 = c * 3;
x_6 = x_4 + x_4;
x_7 = tmp_1 * x_5;
out_1 = x_6 * x_7;
```

- Use mobility to create the priority list
- Available functional units:
  - 1 adder, latency 1 clock cycle
  - 1 multiplier, latency 2 clock cycles
  - 1 ALU, latency 2 clock cycles
- The ALU can perform any kind of arithmetic operations
- a, b, c, and d are input constants

## HLS scheduling

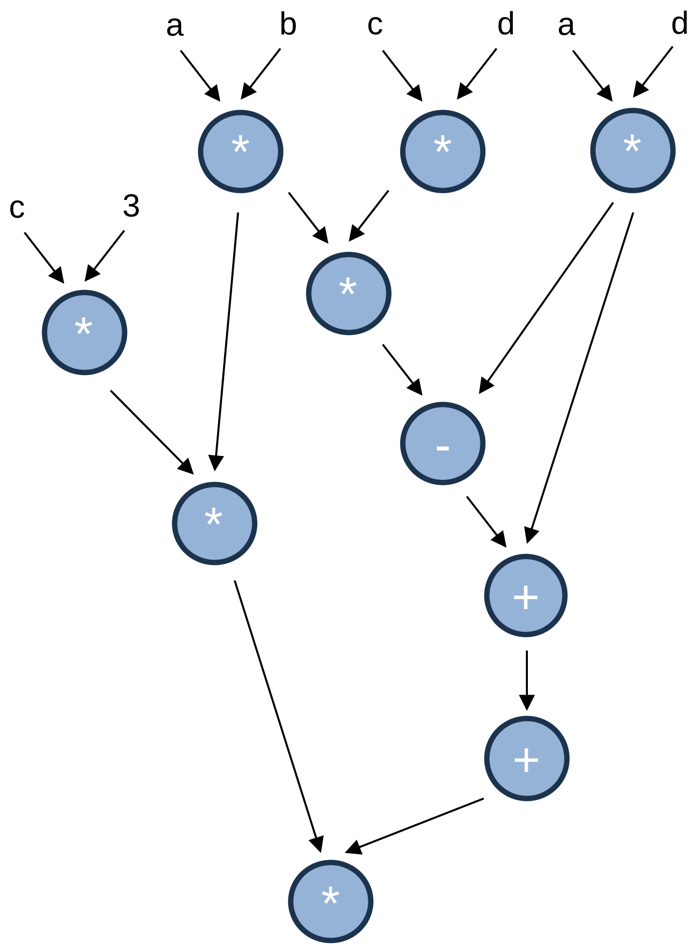
- Building the DFG

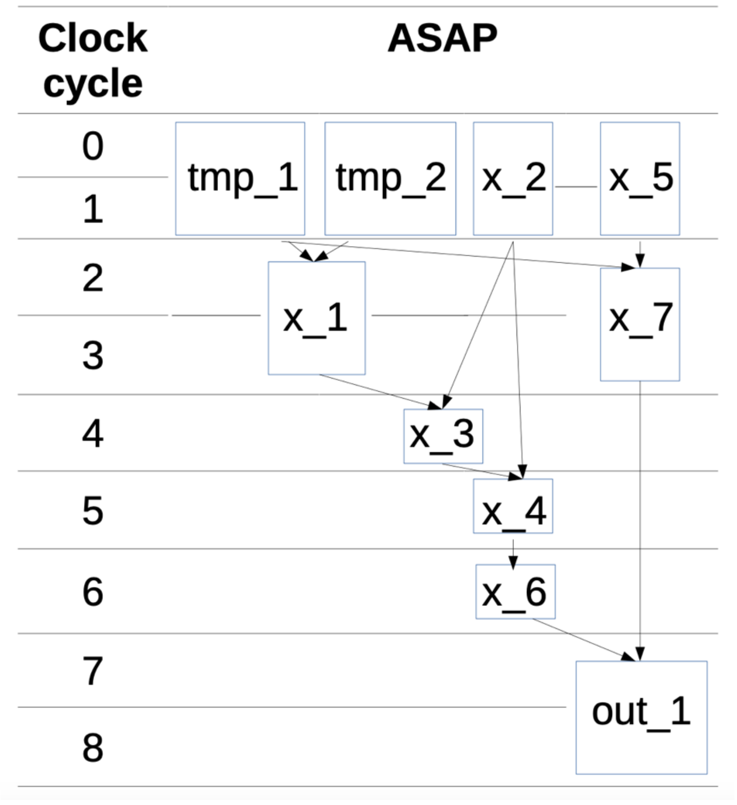
```
tmp_1 = a * b;
tmp_2 = c * d;
x_1 = tmp_1 * tmp_2;
x_2 = a * d;
x_3 = x_2 - x_1;
x_4 = x_2 + x_3;
x_5 = c * 3;
x_6 = x_4 + x_4;
x_7 = tmp_1 * x_5;
out_1 = x_6 * x_7;
```



## HLS scheduling

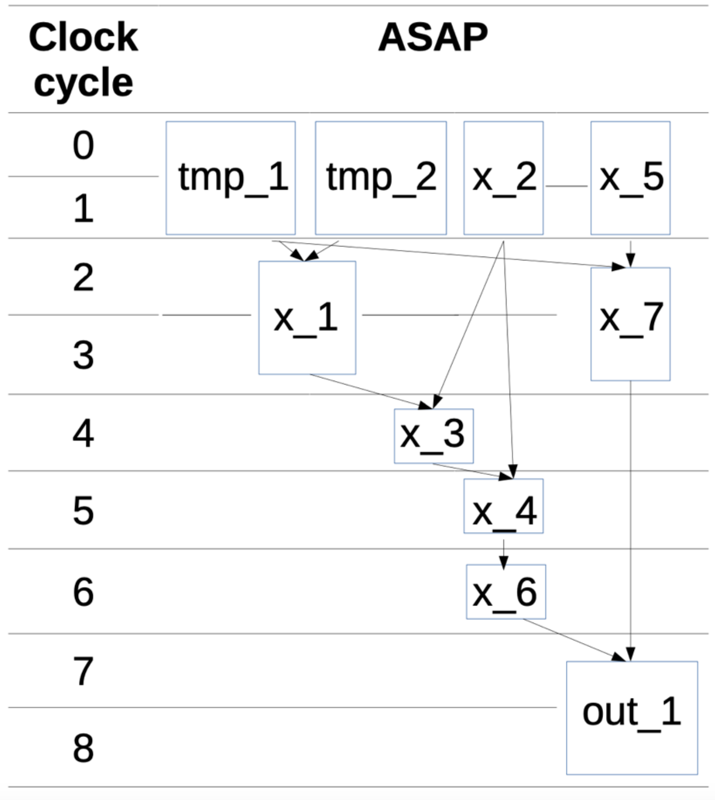
- Priority list = mobility

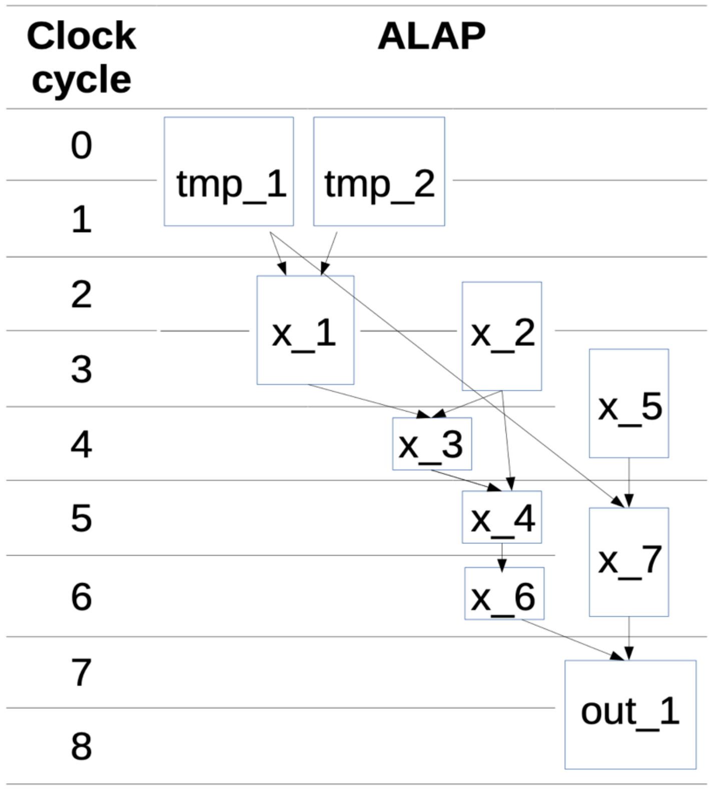




## HLS scheduling

- Priority list = mobility





$\text{Mobility}_{tmp\_1} = 0$
$\text{Mobility}_{tmp\_2} = 0$
$\text{Mobility}_{x\_2} = 2$
$\text{Mobility}_{x\_5} = 3$
$\text{Mobility}_{x\_1} = 0$
$\text{Mobility}_{x\_7} = 3$
$\text{Mobility}_{x\_3} = 0$
$\text{Mobility}_{x\_4} = 0$
$\text{Mobility}_{x\_6} = 0$
$\text{Mobility}_{out\_1} = 0$

## HLS scheduling

- List-based scheduling

| Clock cycle | ADDER | MULTIPLIER | ALU |
|---|---|---|---|
| 1 | | tmp_1 = a * b; | tmp_2 = c * d; |
| 2 | | busy | busy |
| | | | |

Ready list:

```
tmp_1 = a * b;
tmp_2 = c * d;
x_1 = tmp_1 * tmp_2;
x_2 = a * d;
x_3 = x_2 - x_1;
x_4 = x_2 + x_3;
x_5 = c * 3;
x_6 = x_4 + x_4;
x_7 = tmp_1 * x_5;
out_1 = x_6 * x_7;
```

$\text{Mobility}_{tmp\_1} = 0$
$\text{Mobility}_{tmp\_2} = 0$
$\text{Mobility}_{x\_2} = 2$
$\text{Mobility}_{x\_5} = 3$
$\text{Mobility}_{x\_1} = 0$
$\text{Mobility}_{x\_7} = 3$
$\text{Mobility}_{x\_3} = 0$
$\text{Mobility}_{x\_4} = 0$
$\text{Mobility}_{x\_6} = 0$
$\text{Mobility}_{out\_1} = 0$

## HLS scheduling

- List-based scheduling

| Clock cycle | ADDER | MULTIPLIER | ALU |
|---|---|---|---|
| 1 | | tmp_1 = a * b; | tmp_2 = c * d; |
| 2 | | busy | busy |
| 3 | | x_1 = tmp_1 * tmp_2; | x_2 = a * d; |
| 4 | | busy | busy |

Ready list:

```
x_1 = tmp_1 * tmp_2;
x_2 = a * d;
x_3 = x_2 - x_1;
x_4 = x_2 + x_3;
x_5 = c * 3;
x_6 = x_4 + x_4;
x_7 = tmp_1 * x_5;
out_1 = x_6 * x_7;
```

$\text{Mobility}_{x\_2} = 2$
$\text{Mobility}_{x\_5} = 3$
$\text{Mobility}_{x\_1} = 0$
$\text{Mobility}_{x\_7} = 3$
$\text{Mobility}_{x\_3} = 0$
$\text{Mobility}_{x\_4} = 0$
$\text{Mobility}_{x\_6} = 0$
$\text{Mobility}_{out\_1} = 0$

## HLS scheduling

- List-based scheduling

| Clock cycle | ADDER | MULTIPLIER | ALU |
|---|---|---|---|
| 1 | | tmp_1 = a * b; | tmp_2 = c * d; |
| 2 | | busy | busy |
| 3 | | x_1 = tmp_1 * tmp_2; | x_2 = a * d; |
| 4 | | busy | busy |
| 5 | x_3 = x_2 - x_1; | x_5 = c * 3; | |

Ready list:

```
x_3 = x_2 - x_1;
x_4 = x_2 + x_3;
x_5 = c * 3;
x_6 = x_4 + x_4;
x_7 = tmp_1 * x_5;
out_1 = x_6 * x_7;
```

$\boxed{\text{Mobility}_{x\_5} = 3}$

$\text{Mobility}_{x\_7} = 3$

$\boxed{\text{Mobility}_{x\_3} = 0}$

$\text{Mobility}_{x\_4} = 0$

$\text{Mobility}_{x\_6} = 0$

$\text{Mobility}_{out\_1} = 0$

## HLS scheduling

- List-based scheduling

| Clock cycle | ADDER | MULTIPLIER | ALU |
|---|---|---|---|
| 1 | | tmp_1 = a * b; | tmp_2 = c * d; |
| 2 | | busy | busy |
| 3 | | x_1 = tmp_1 * tmp_2; | x_2 = a * d; |
| 4 | | busy | busy |
| 5 | x_3 = x_2 - x_1; | x_5 = c * 3; | |
| 6 | x_4 = x_2 + x_3; | busy | |

Ready list:

```
x_4 = x_2 + x_3;
x_6 = x_4 + x_4;
x_7 = tmp_1 * x_5;
out_1 = x_6 * x_7;
```

$\text{Mobility}_{x\_7} = 3$

$\boxed{\text{Mobility}_{x\_4} = 0}$

$\text{Mobility}_{x\_6} = 0$

$\text{Mobility}_{out\_1} = 0$

## HLS scheduling

- List-based scheduling

| Clock cycle | ADDER | MULTIPLIER | ALU |
|---|---|---|---|
| 1 | | tmp_1 = a * b; | tmp_2 = c * d; |
| 2 | | busy | busy |
| 3 | | x_1 = tmp_1 * tmp_2; | x_2 = a * d; |
| 4 | | busy | busy |
| 5 | x_3 = x_2 - x_1; | x_5 = c * 3; | |
| 6 | x_4 = x_2 + x_3; | busy | |
| 7 | x_6 = x_4 + x_4; | x_7 = tmp_1 * x_5; | |
| 8 | | busy | |

Ready list:

```
x_6 = x_4 + x_4;
x_7 = tmp_1 * x_5;
out_1 = x_6 * x_7;
```

$\text{Mobility}_{x\_7} = 3$

$\text{Mobility}_{x\_6} = 0$

$\text{Mobility}_{out\_1} = 0$

## HLS scheduling

- List-based scheduling

| Clock cycle | ADDER | MULTIPLIER | ALU |
|---|---|---|---|
| 1 | | tmp_1 = a * b; | tmp_2 = c * d; |
| 2 | | busy | busy |
| 3 | | x_1 = tmp_1 * tmp_2; | x_2 = a * d; |
| 4 | | busy | busy |
| 5 | x_3 = x_2 - x_1; | x_5 = c * 3; | |
| 6 | x_4 = x_2 + x_3; | busy | |
| 7 | x_6 = x_4 + x_4; | x_7 = tmp_1 * x_5; | |
| 8 | | busy | |
| 9 | | out_1 = x_6 * x_7; | |
| 10 | | busy | |

Ready list: `out_1 = x_6 * x_7;`

$\boxed{\text{Mobility}_{out\_1} = 0}$

## Using Bambu HLS

https://colab.research.google.com/drive/1muPx2tSure0GmyljW4q5G0J6QdHclXZE?usp=share_link

## Credits & References

- Stanford course CS149, Parallel Computing
- https://halide-lang.org/
  - Links to source code, publications, tutorials, all sorts of useful material
- "Digital & Computational Photography", MIT course by Frédo Durand (lectures 14 to 17) https://stellar.mit.edu/S/course/6/sp15/6.815/materials.html

## Credits & References

- M. Fingeroff, “High-Level Synthesis Blue Book”
  - https://cse.usf.edu/~haozheng/teach/cda4253/doc/hls/hls_bluebook_uv.pdf
- Bambu tutorial from DATE 2022 (somewhat outdated, but still useful)
- In-depth Vitis HLS user guide and tutorials
- J. Cong et al., “FPGA HLS Today: Successes, Challenges, and Opportunities” https://doi.org/10.1145/3530775
- G. Gozzi et al., “SPARTA: High-Level Synthesis of Parallel Multi-Threaded Accelerators” https://doi.org/10.1145/3677035