Synthesizing Code for GPGPUs from Abstract Formal Models

Gabriel Hjort Blindell
Christian Menne
Ingo Sander

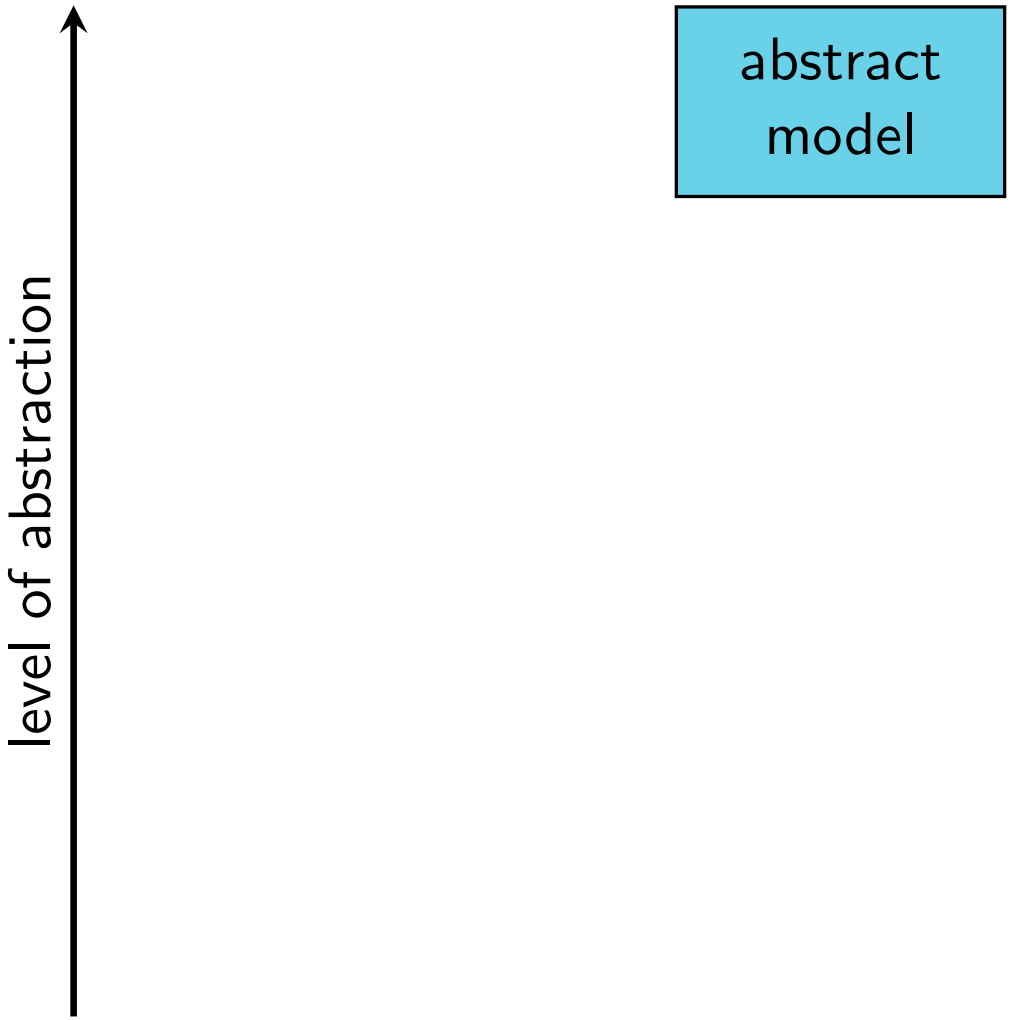
KTH Royal Institute of Technology, Sweden

November 20, 2014
SCS Seminar

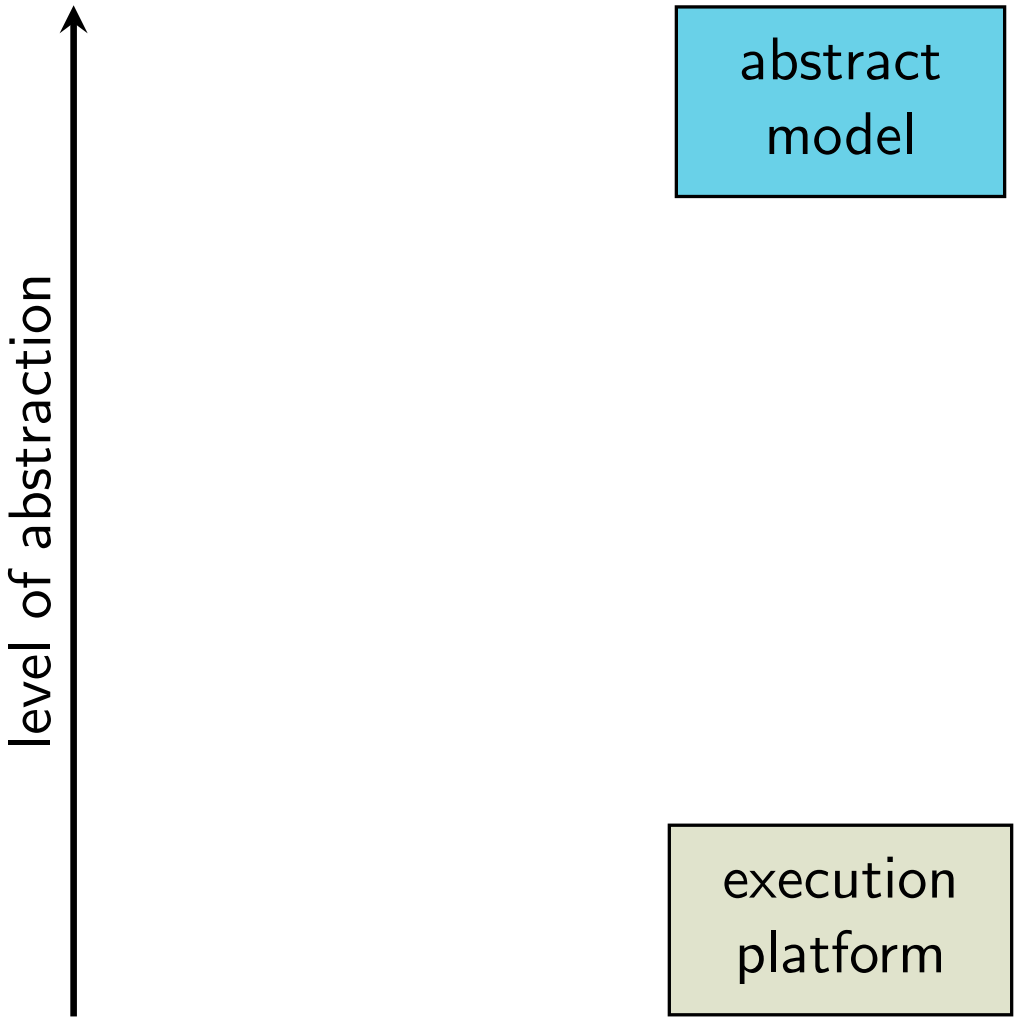
Paper

Hjort Blindell, G., Menne, C., Sander, I. "Synthesizing Code for GPGPUs from Abstract Formal Models." Forum on specification & Design Languages (FDL 2014). Munich, Germany, October 14–16, 2014.

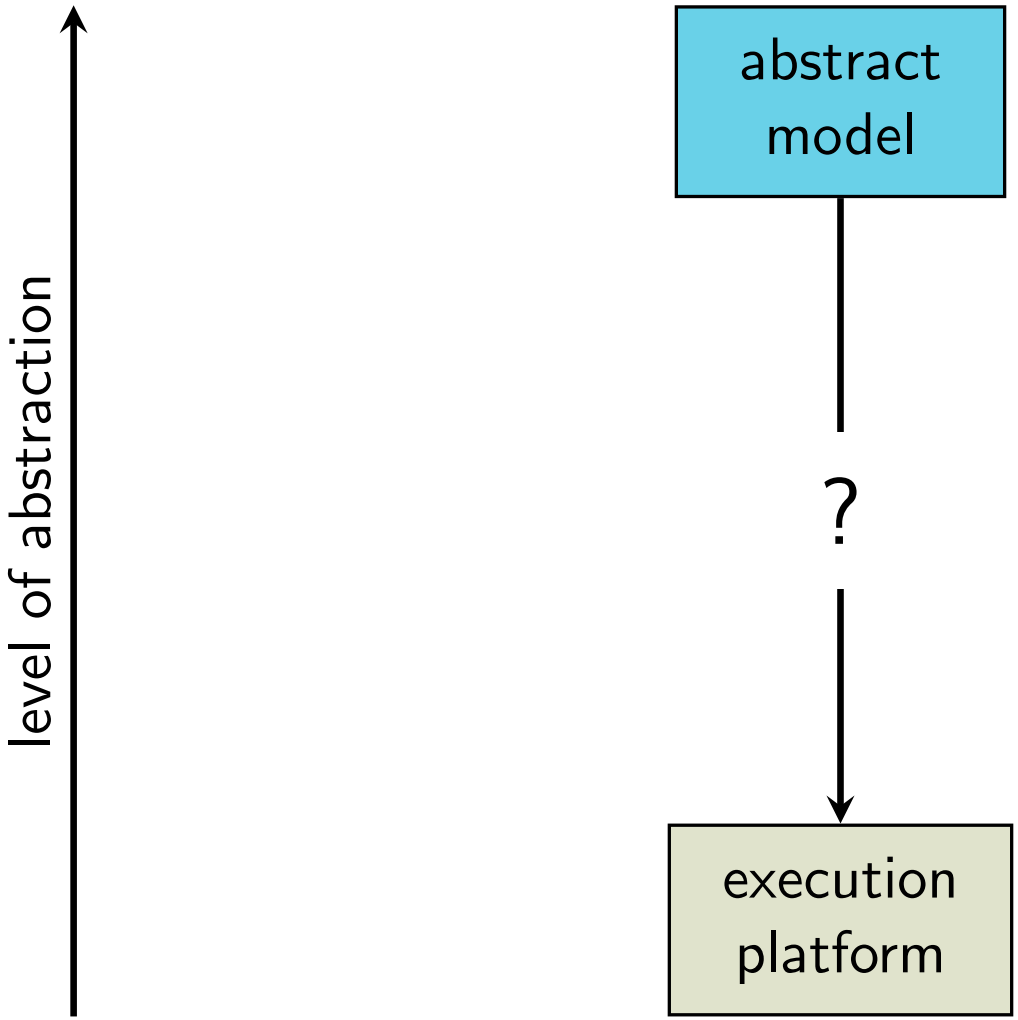
Ideal: Want to Model at a High Level of Abstraction



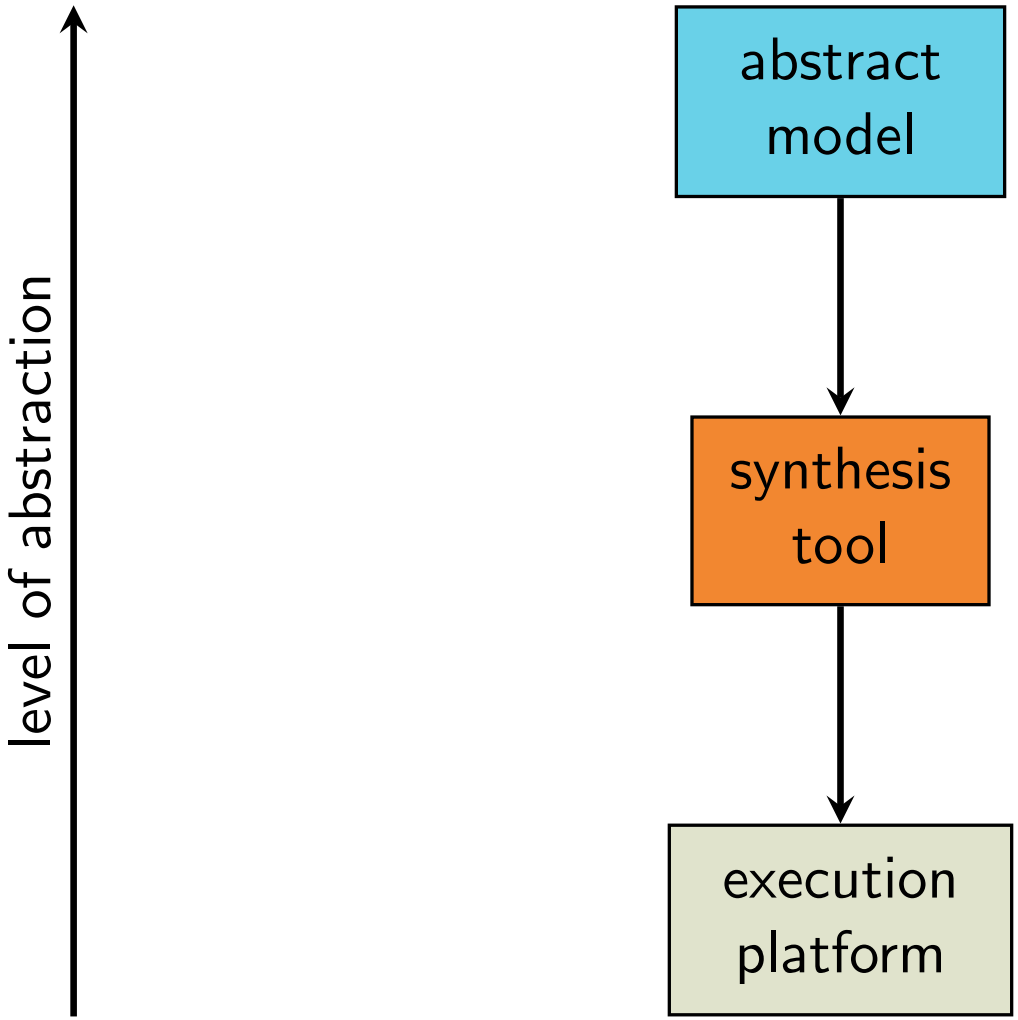
Reality: Have to Implement at a Low Level of Abstraction



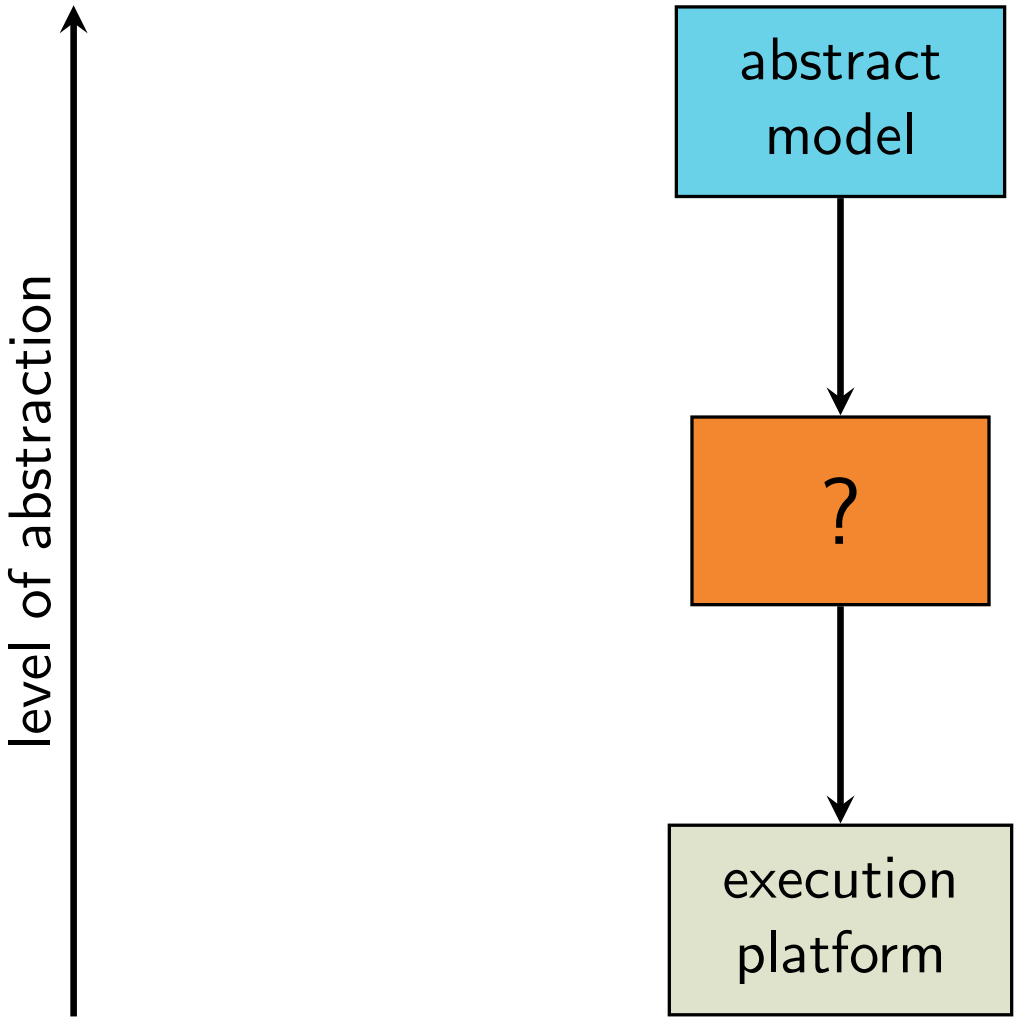
Problem: How to Bridge the Gap?



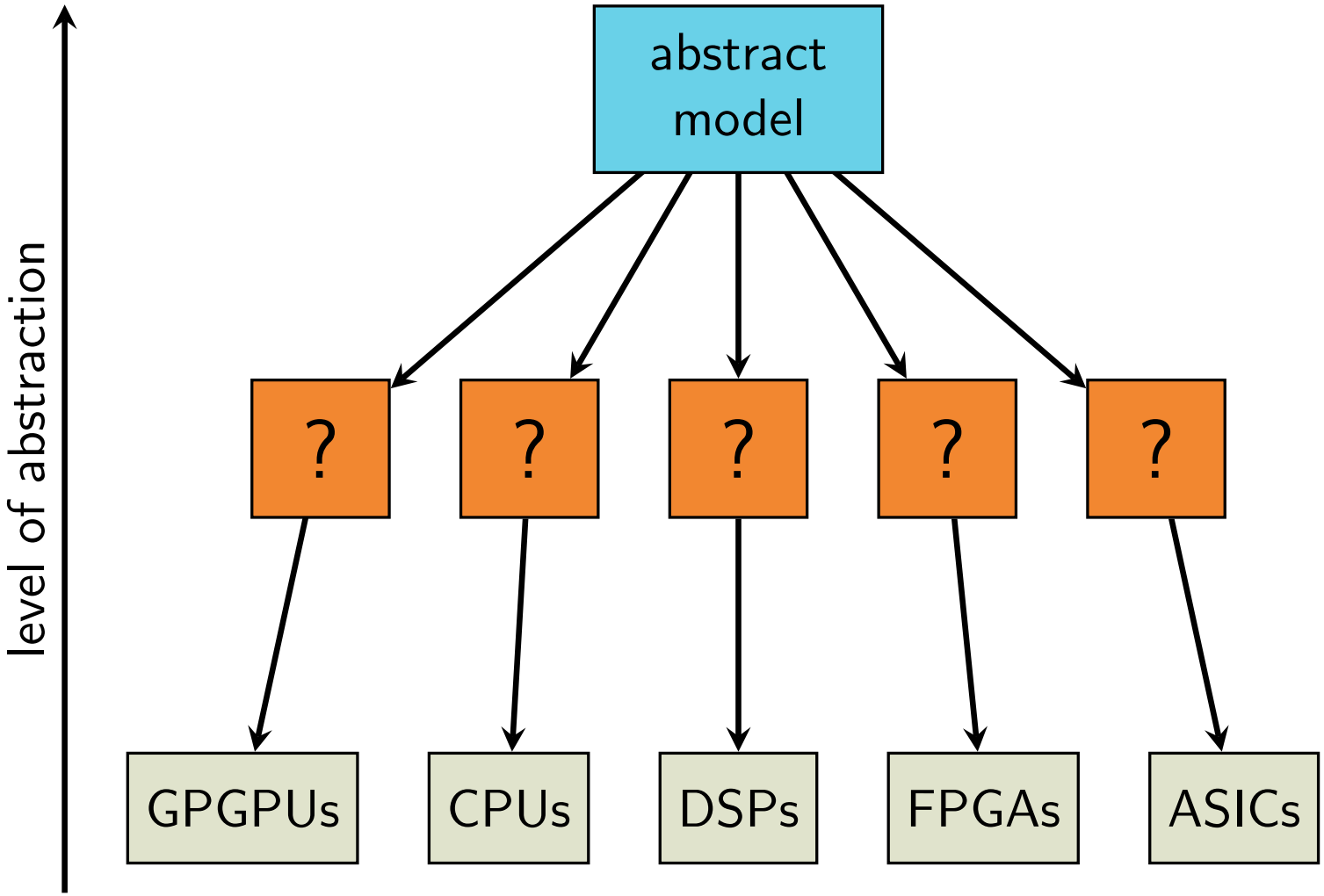
Solution: Use Automated Synthesis Tools



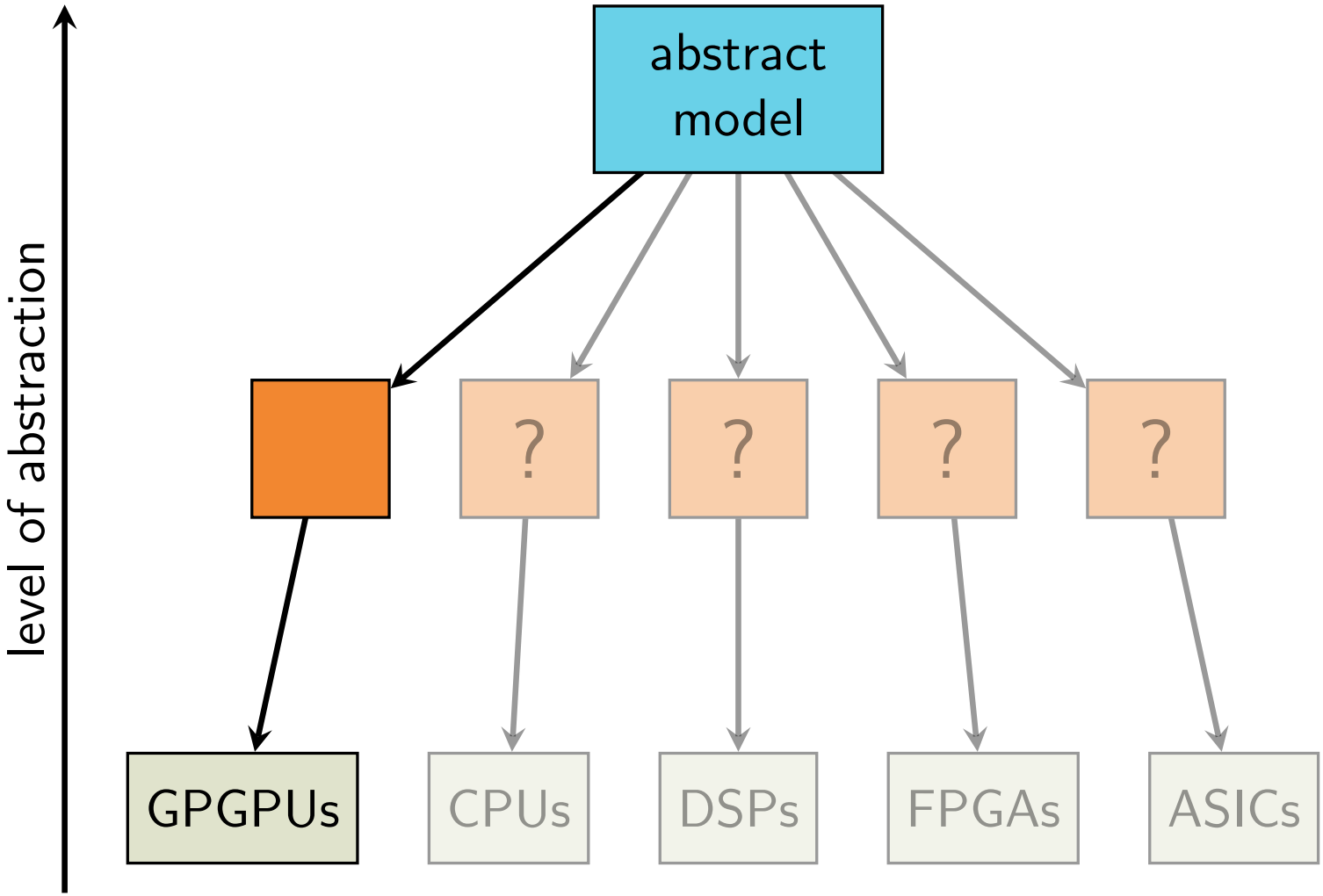
New Problem: How to Build Such a Tool?



New Problem: Different Challenges for Different Platforms



This Talk: A Synthesis Tool for GPGPUs



Outline

- Background
  - GPGPUs
  - ForSyDe
- Our ideas and synthesis tool (`f2cc`)
- Experiments
- Summary

What Are GPGPUs?

What Are GPGPUs?

- *General-Purpose Graphics Processing Unit*

What Are GPGPUs?

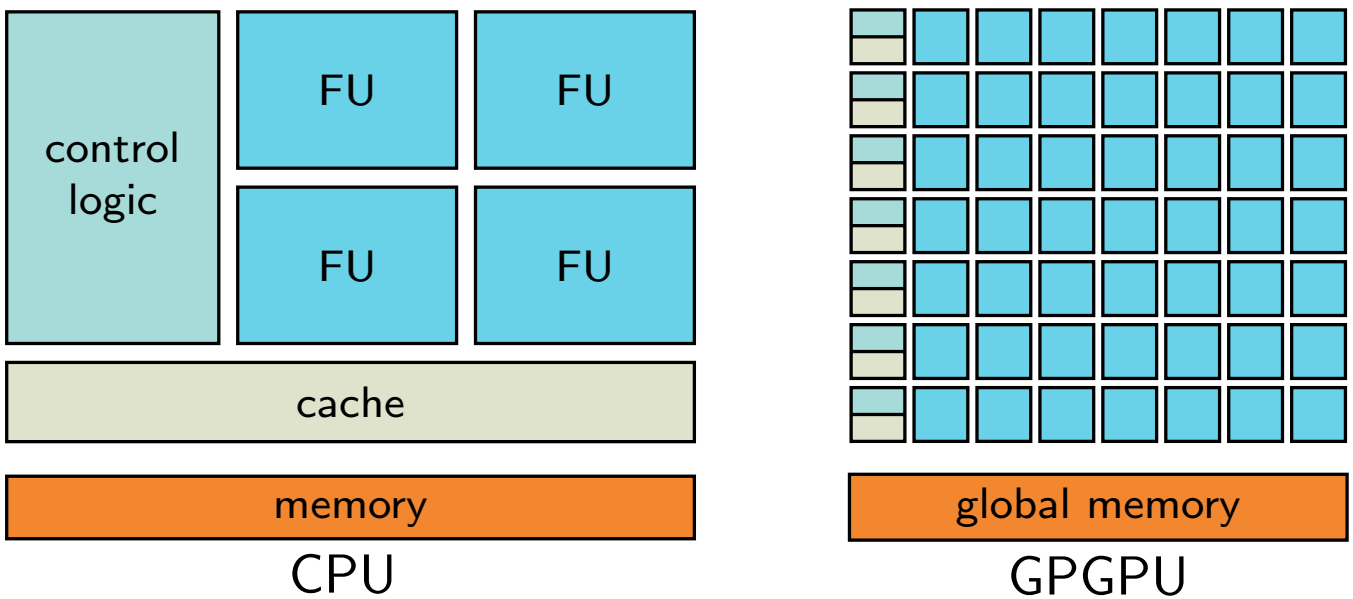
- *General-Purpose Graphics Processing Unit*
- Massively parallel, throughput-oriented platform

What Are GPGPUs?

- *General-Purpose Graphics Processing Unit*
- Massively parallel, throughput-oriented platform
- Can yield tremendous speedup for data-parallel programs

What Are GPGPUs?

- *General-Purpose Graphics Processing Unit*
- Massively parallel, throughput-oriented platform
- Can yield tremendous speedup for data-parallel programs
- Comparison between CPUs and GPGPUs:



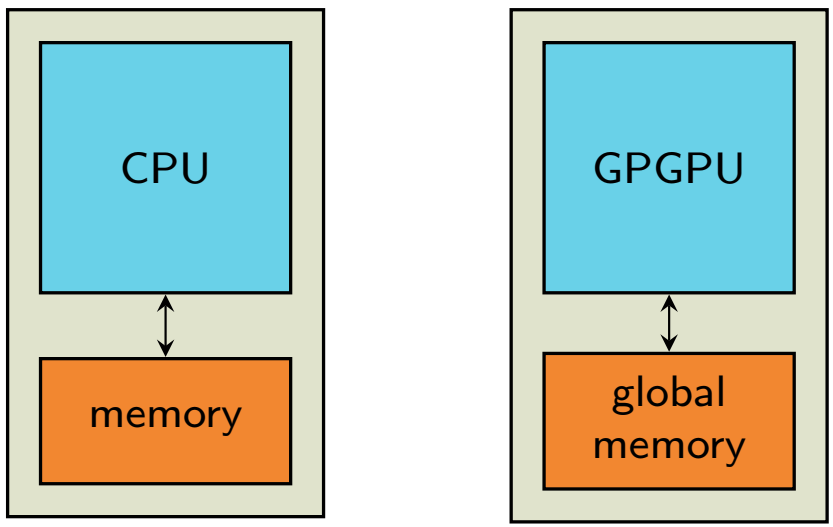
How to Use GPGPUs?

How to Use GPGPUs?

- Programmed using C dialect (here assuming *CUDA C*)

How to Use GPGPUs?

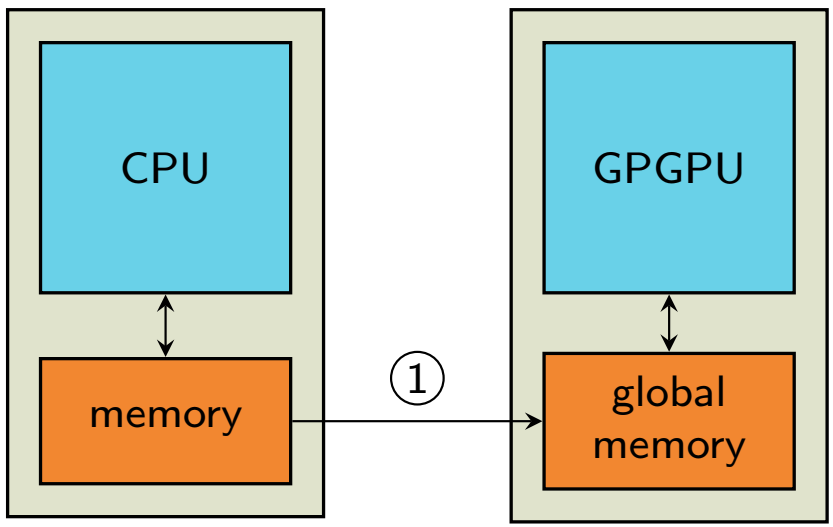
- Programmed using C dialect (here assuming *CUDA C*)
- Treated as an *accelerator*



How to Use GPGPUs?

- Programmed using C dialect (here assuming *CUDA C*)
- Treated as an *accelerator*

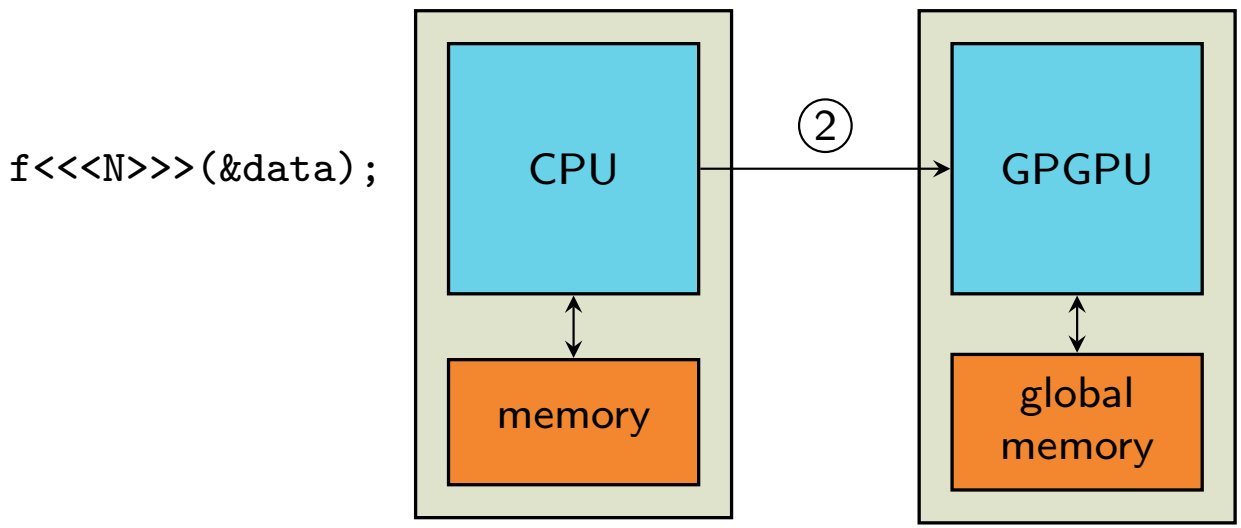
```
cudaMalloc(...);
cudaMemcpy(&data);
```



Copy input data

How to Use GPGPUs?

- Programmed using C dialect (here assuming *CUDA C*)
- Treated as an *accelerator*

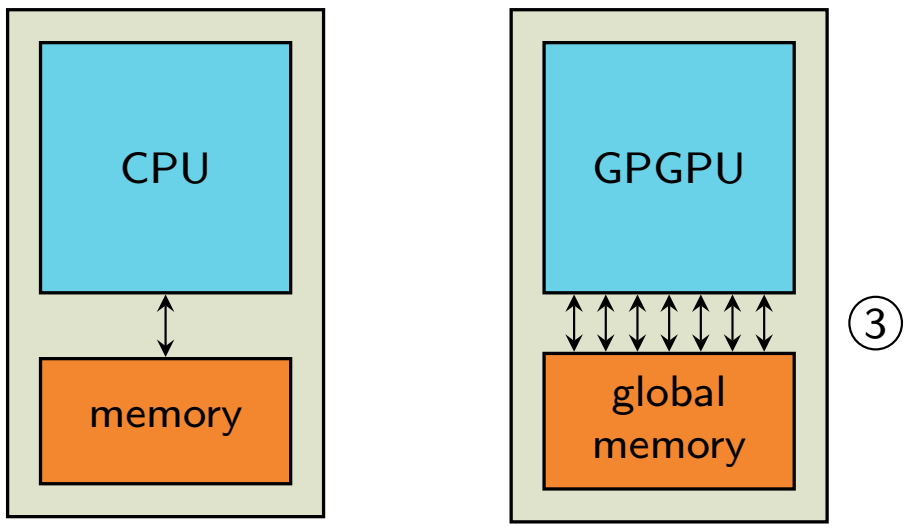


Tell GPGPU to execute function f on input data, using N threads

How to Use GPGPUs?

- Programmed using C dialect (here assuming *CUDA C*)
- Treated as an *accelerator*

```
while (1);
```

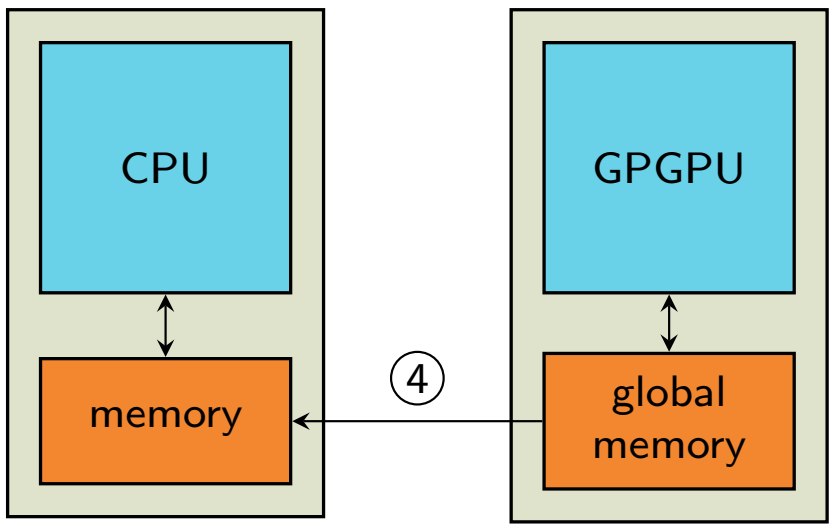


Wait until all threads have finished

How to Use GPGPUs?

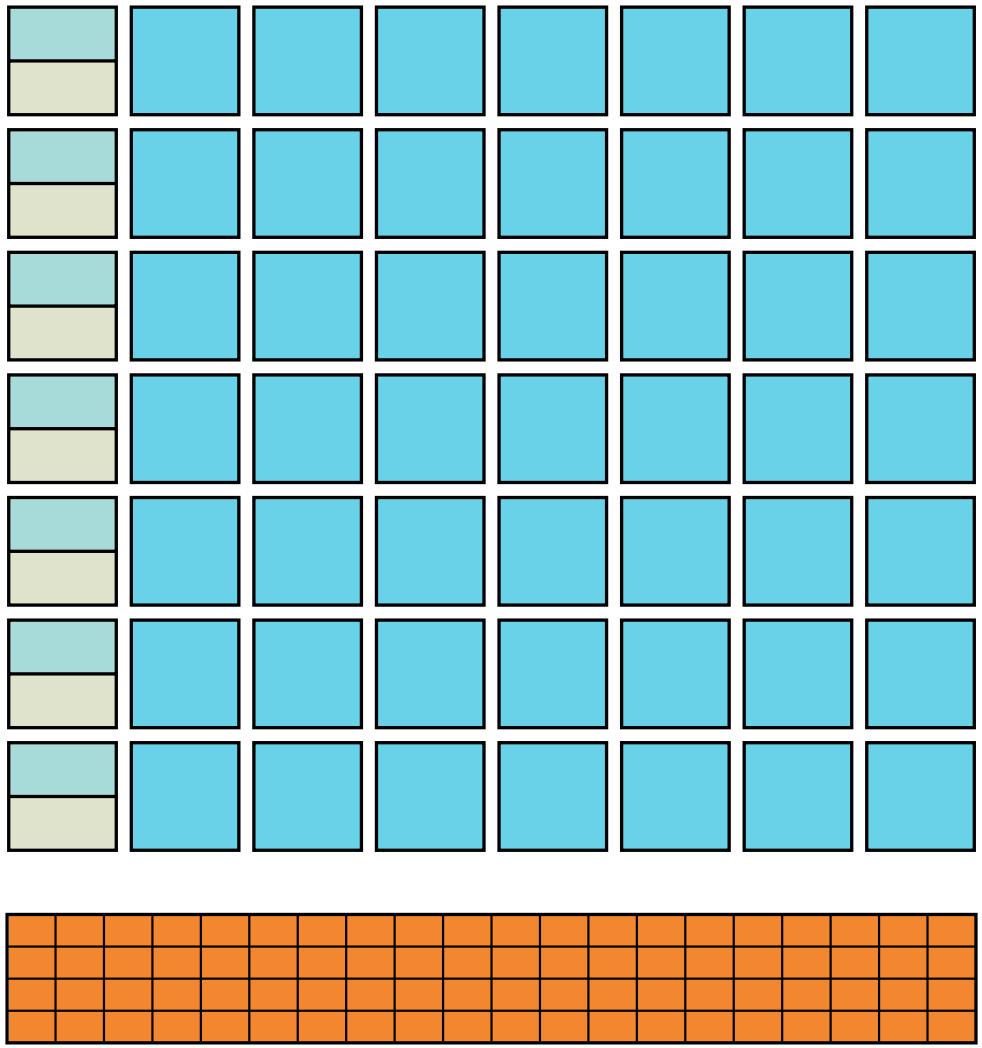
- Programmed using C dialect (here assuming *CUDA C*)
- Treated as an *accelerator*

```
cudaMemcpy(&res);
cudaFree(...);
```

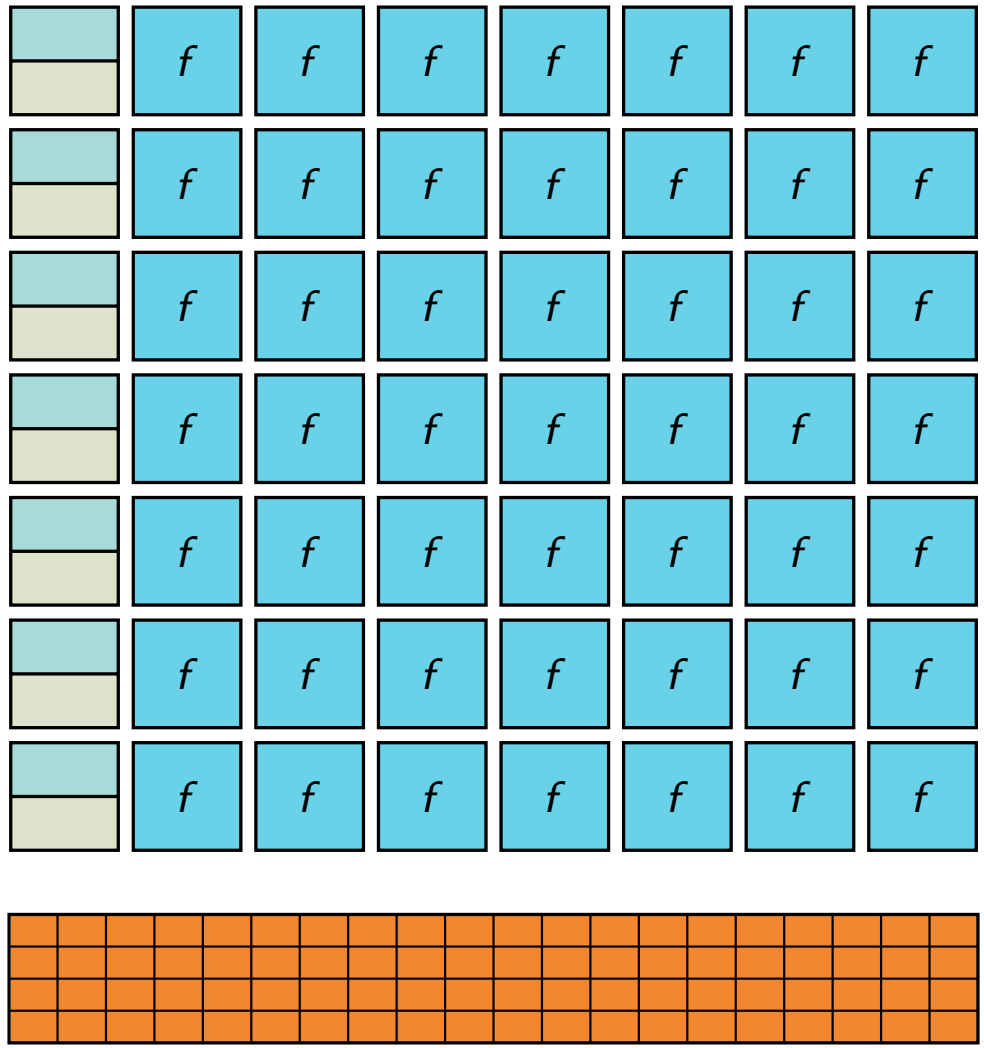


Copy result

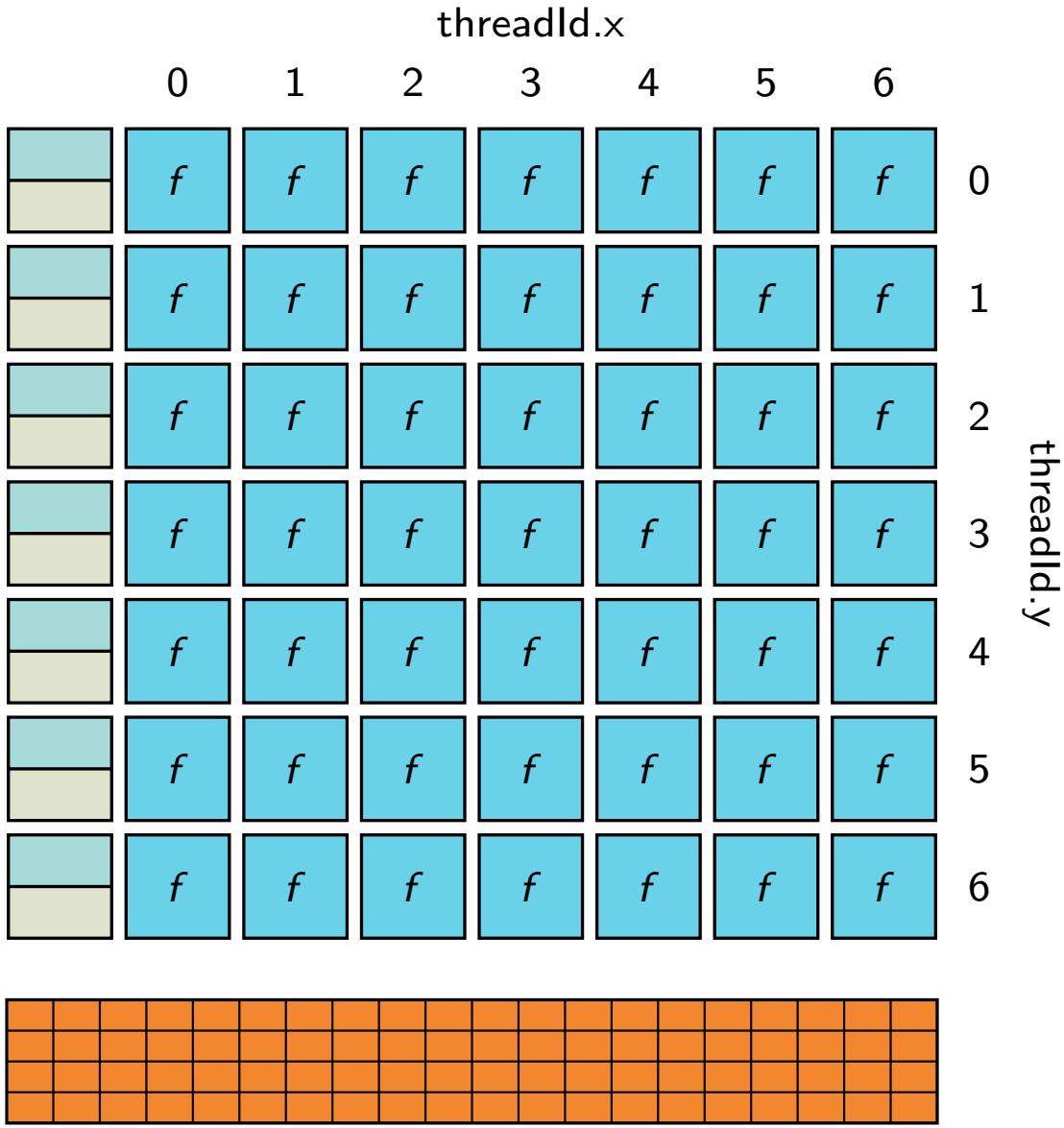
Inside the GPGPU During Execution



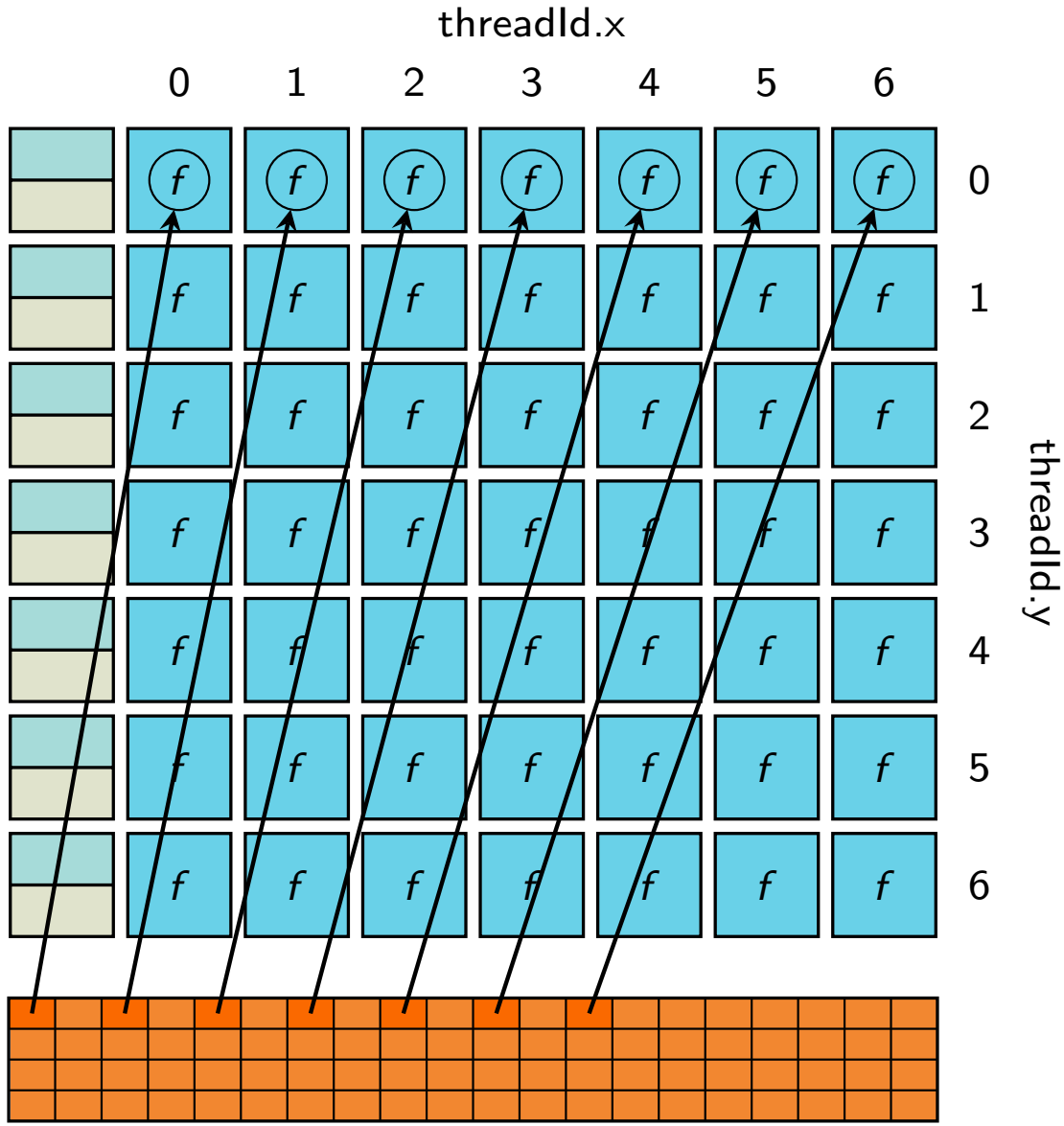
Every Thread Executes the Same f



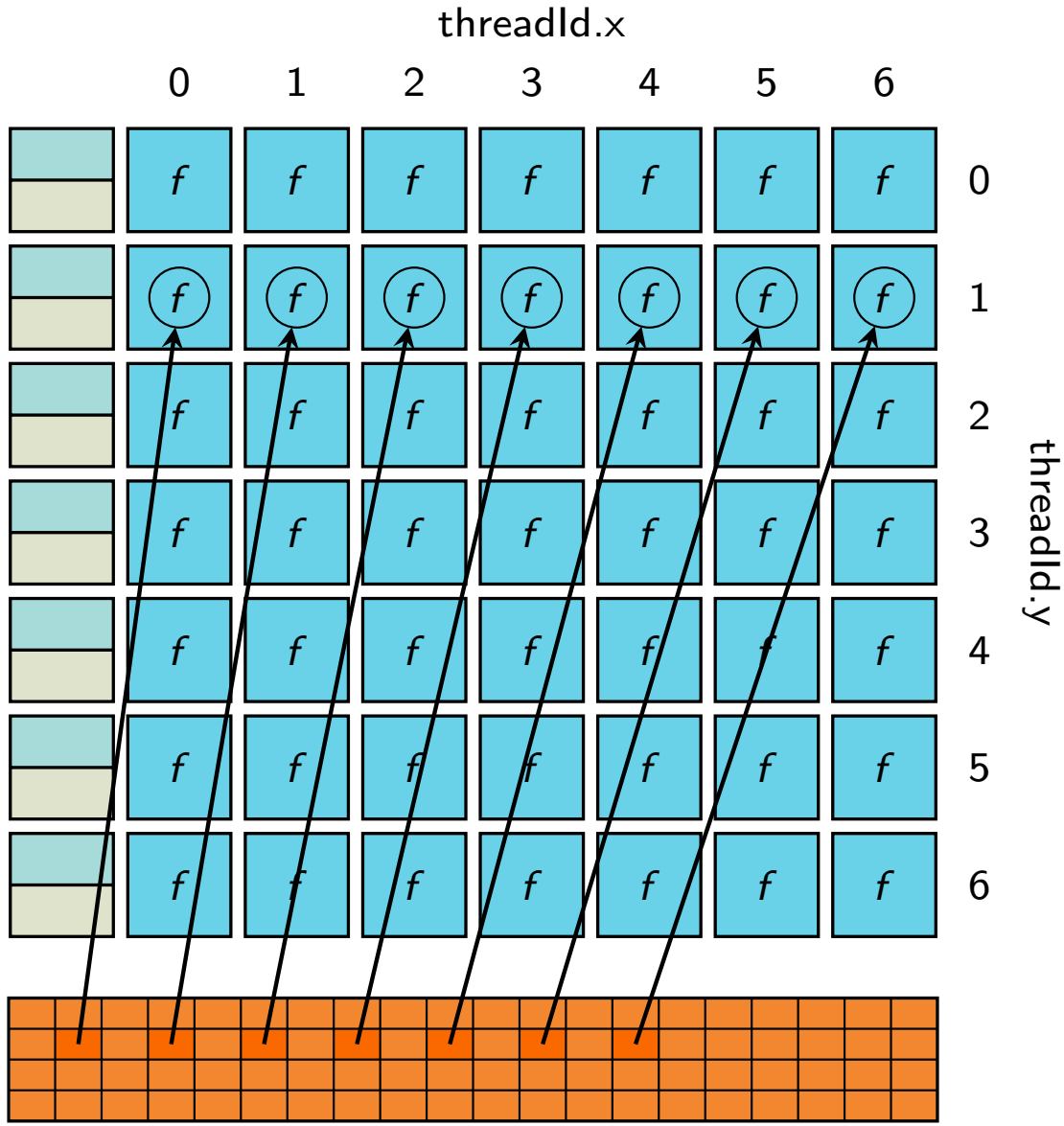
Each Thread Has a Unique *Thread ID*



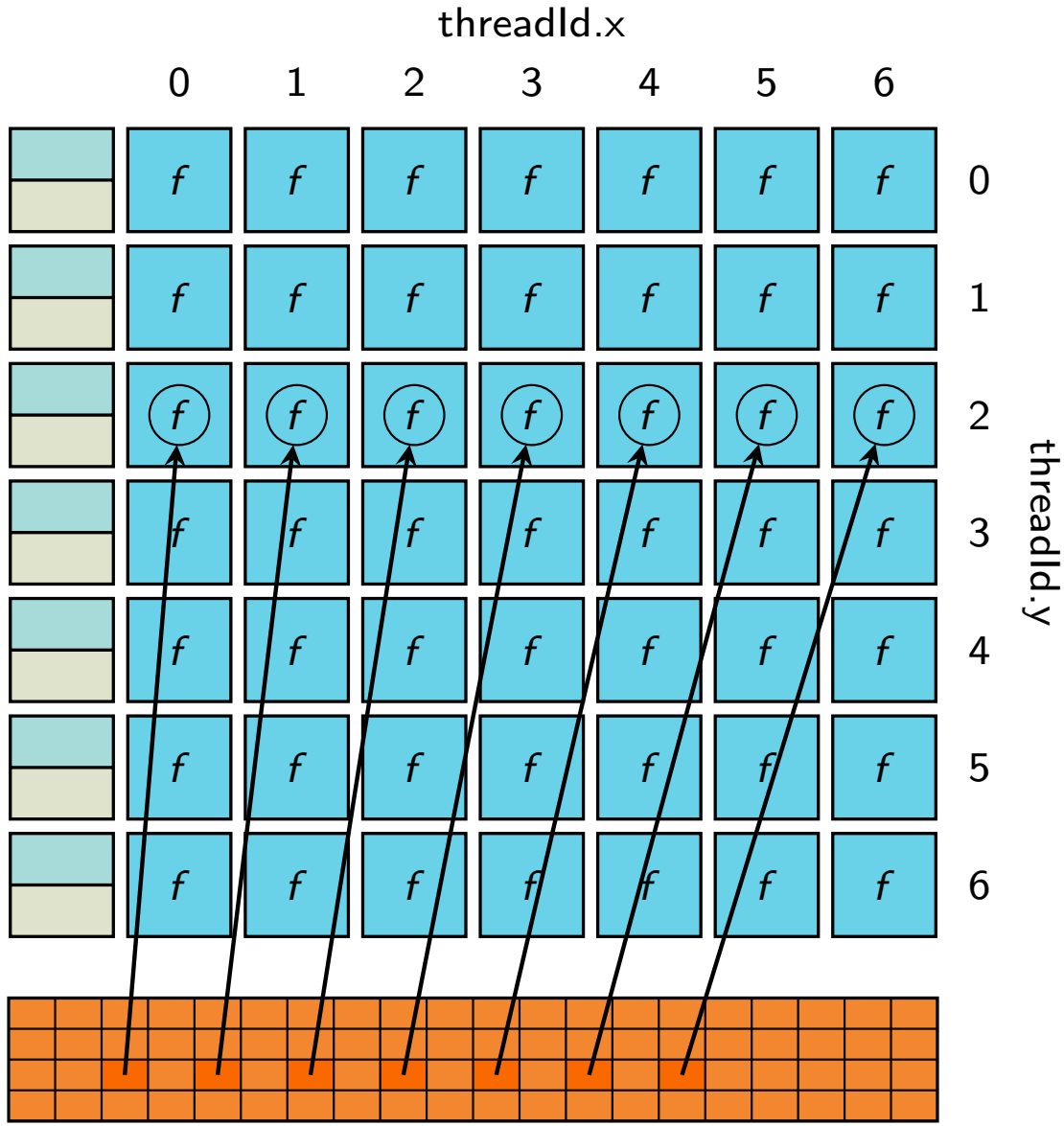
f Uses Thread ID to Determine What Data to Read



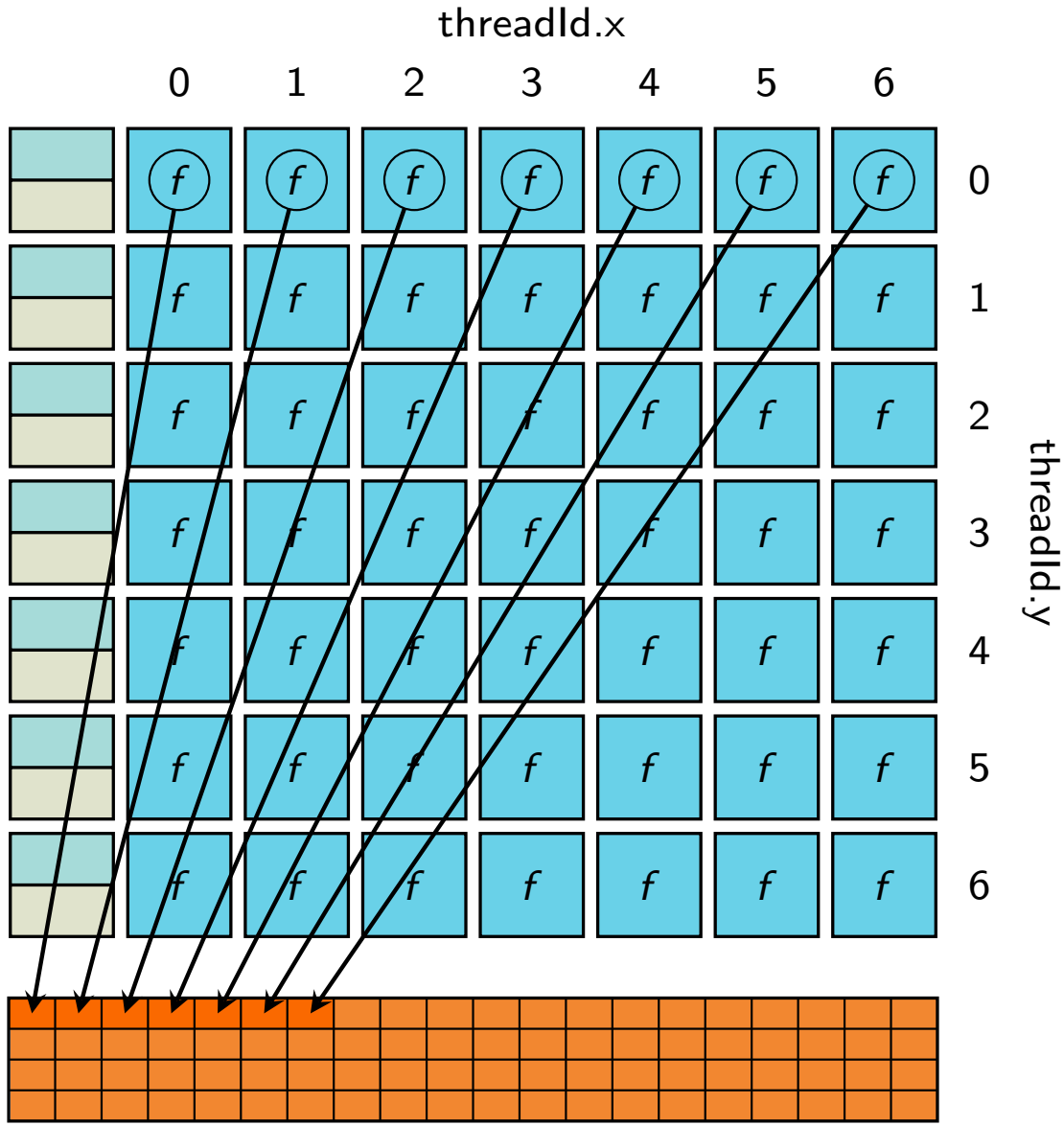
f Uses Thread ID to Determine What Data to Read



f Uses Thread ID to Determine What Data to Read



f Uses Thread ID to Determine Where to Write Results



Requirements for Optimal Performance

Requirements for Optimal Performance

- Abundance of data parallelism to offset GPGPU overhead (due mainly for data copying)

Requirements for Optimal Performance

- Abundance of data parallelism to offset GPGPU overhead (due mainly for data copying)
- High Computation-to-Global Memory Traffic Ratio

Requirements for Optimal Performance

- Abundance of data parallelism to offset GPGPU overhead (due mainly for data copying)
- High Computation-to-Global Memory Traffic Ratio
  - Often requires efficient use of various resources (like *shared memory*)

Requirements for Optimal Performance

- Abundance of data parallelism to offset GPGPU overhead (due mainly for data copying)
- High Computation-to-Global Memory Traffic Ratio
  - Often requires efficient use of various resources (like *shared memory*)
- No resource over-use

GPGPUs are Powerful, but Difficult to Program

GPGPUs are Powerful, but Difficult to Program

- Complex data indexing schemes

GPGPUs are Powerful, but Difficult to Program

- Complex data indexing schemes
- Performance depends on many interconnected factors

What Is ForSyDe?

What Is ForSyDe?

- *Formal System Design*

What Is ForSyDe?

- *Formal System Design*
- A formal modeling methodology

What Is ForSyDe?

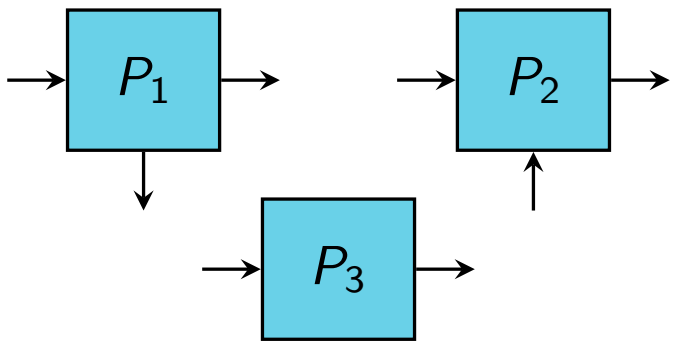
- *Formal System Design*
- A formal modeling methodology
  - Uses the theory of *Models of Computation (MoCs)*

What Is ForSyDe?

- *Formal System Design*
- A formal modeling methodology
  - Uses the theory of *Models of Computation (MoCs)*
  - Captures behavior of heterogeneous embedded systems as *ForSyDe models*

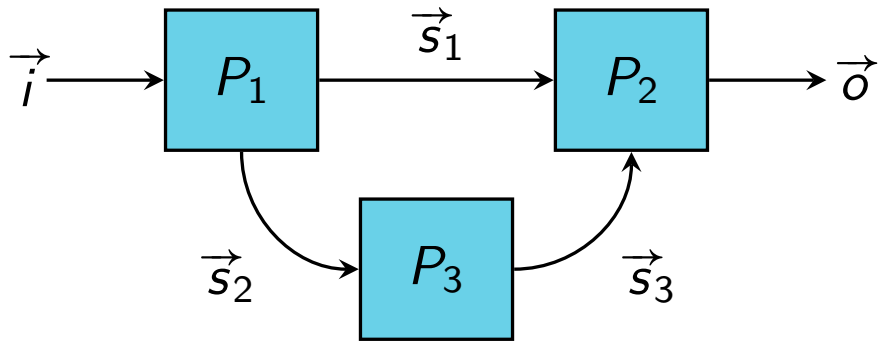
What Is a ForSyDe Model?

What Is a ForSyDe Model?



A ForSyDe model is a *concurrent network* of *processes* ...

What Is a ForSyDe Model?



. . . that communicate via *signals*.

What Is a Process Constructor?

What Is a Process Constructor?

process constructor

A process constructor is a *template* . . .

What Is a Process Constructor?

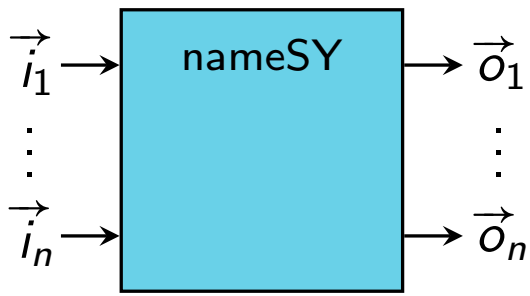
process constructor

based on the
synchronous MoC

. . . that is *based* on a specific model of computation, . . .

What Is a Process Constructor?

process constructor

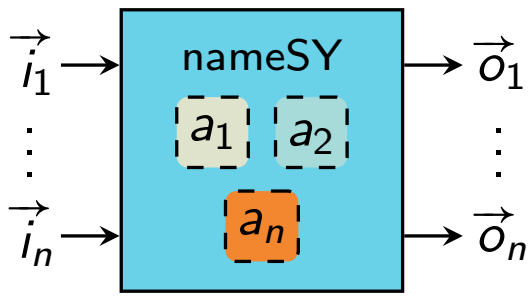


based on the
synchronous MoC

. . . has a number of declared *input* and *output signals*, . . .

What Is a Process Constructor?

process constructor



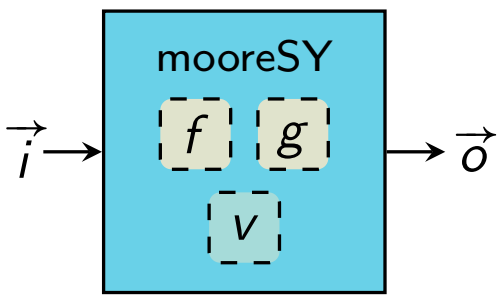
based on the
synchronous MoC

. . . and takes zero or more *side effect-free arguments*.

Creating a Process

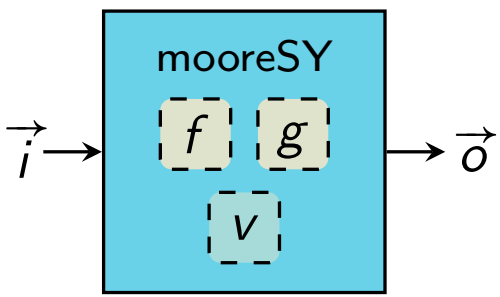
Creating a Process

process constructor



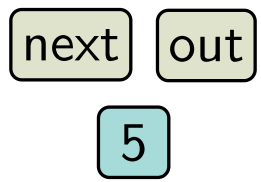
Creating a Process

process constructor

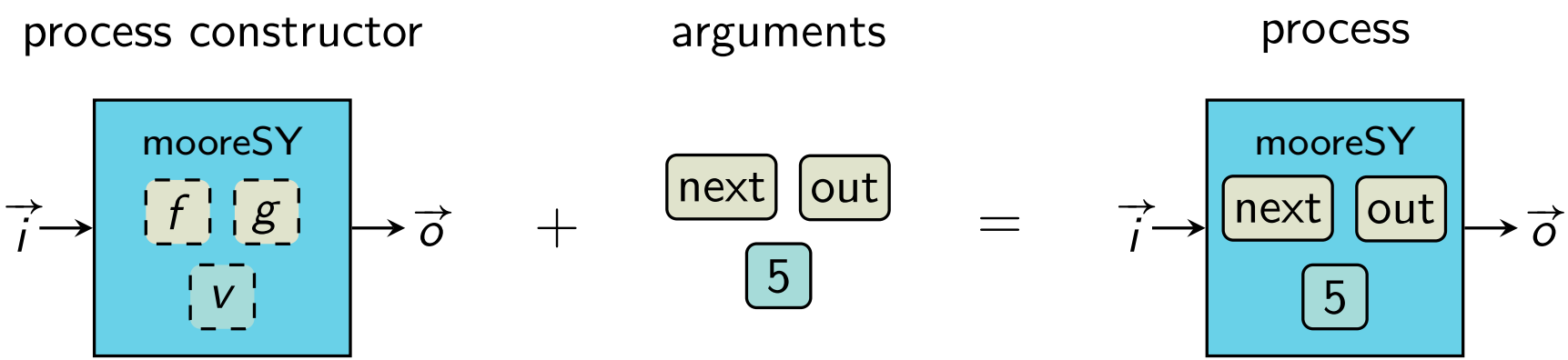


+

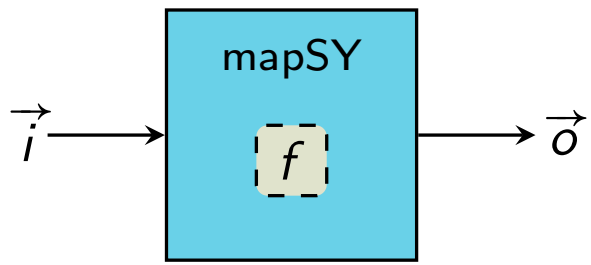
arguments



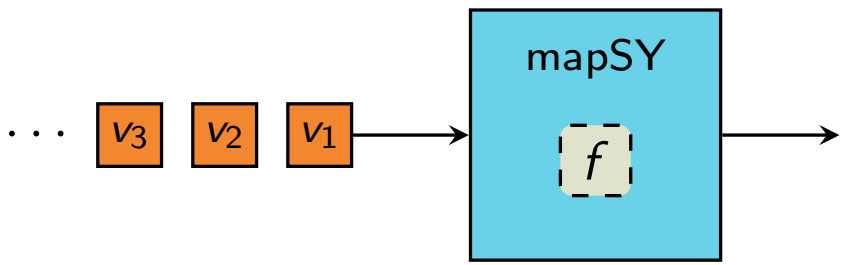
Creating a Process



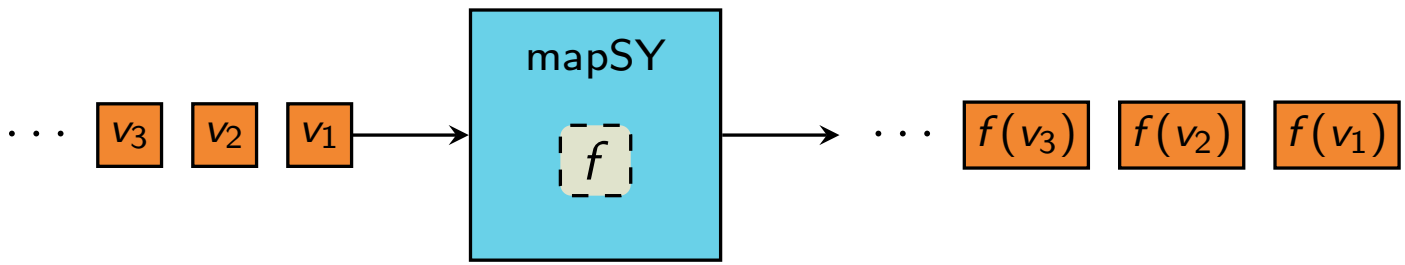
The MapSY Process Constructor



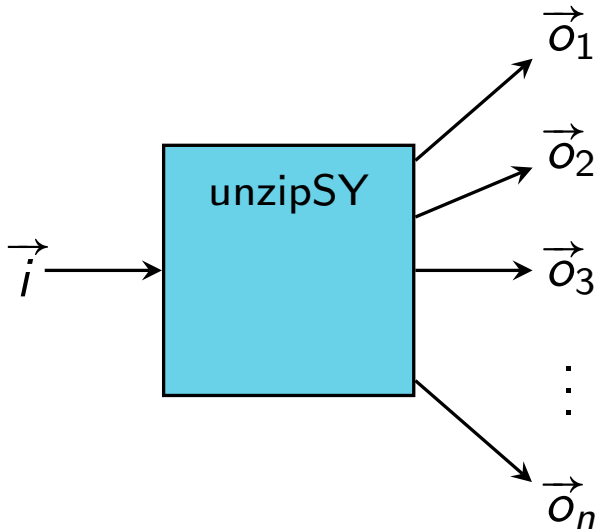
The MapSY Process Constructor



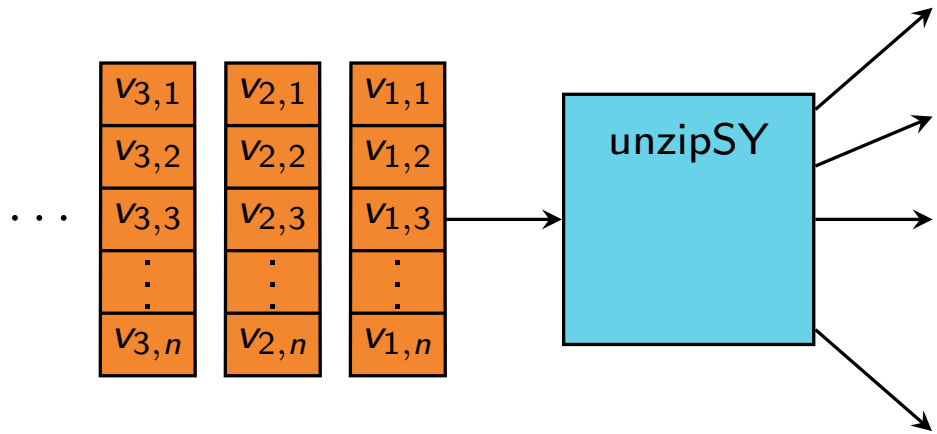
The MapSY Process Constructor



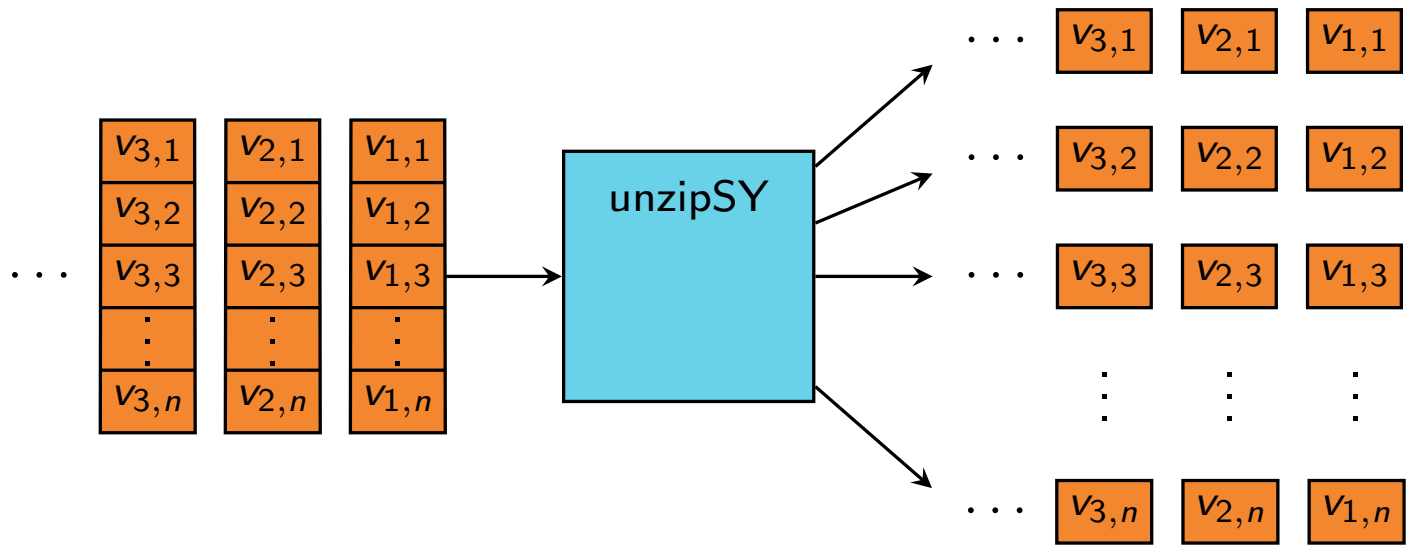
The UnzipSY Process Constructor



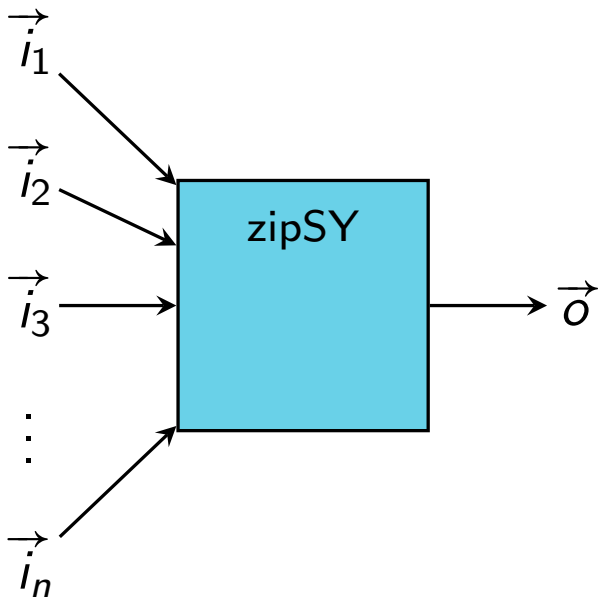
The UnzipSY Process Constructor



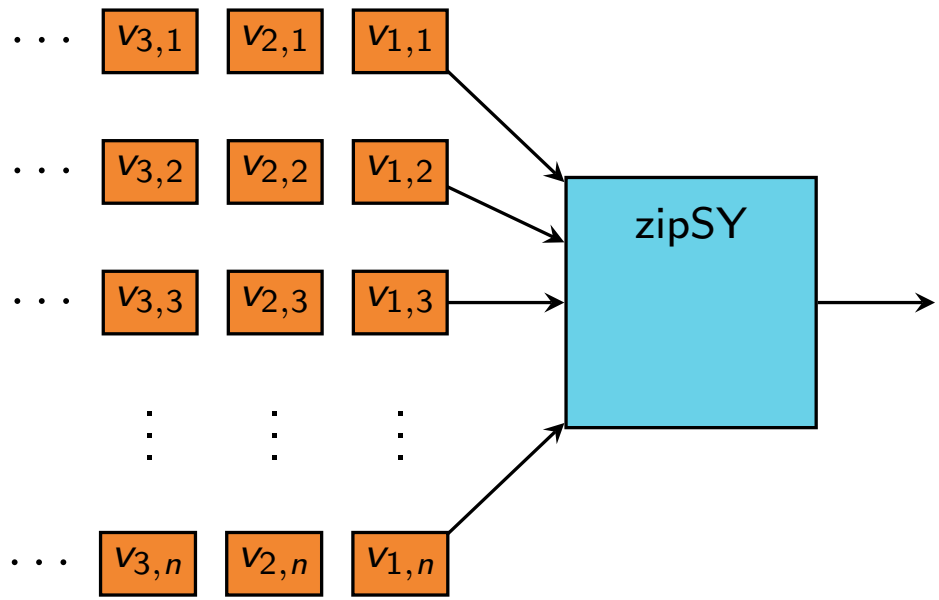
The UnzipSY Process Constructor



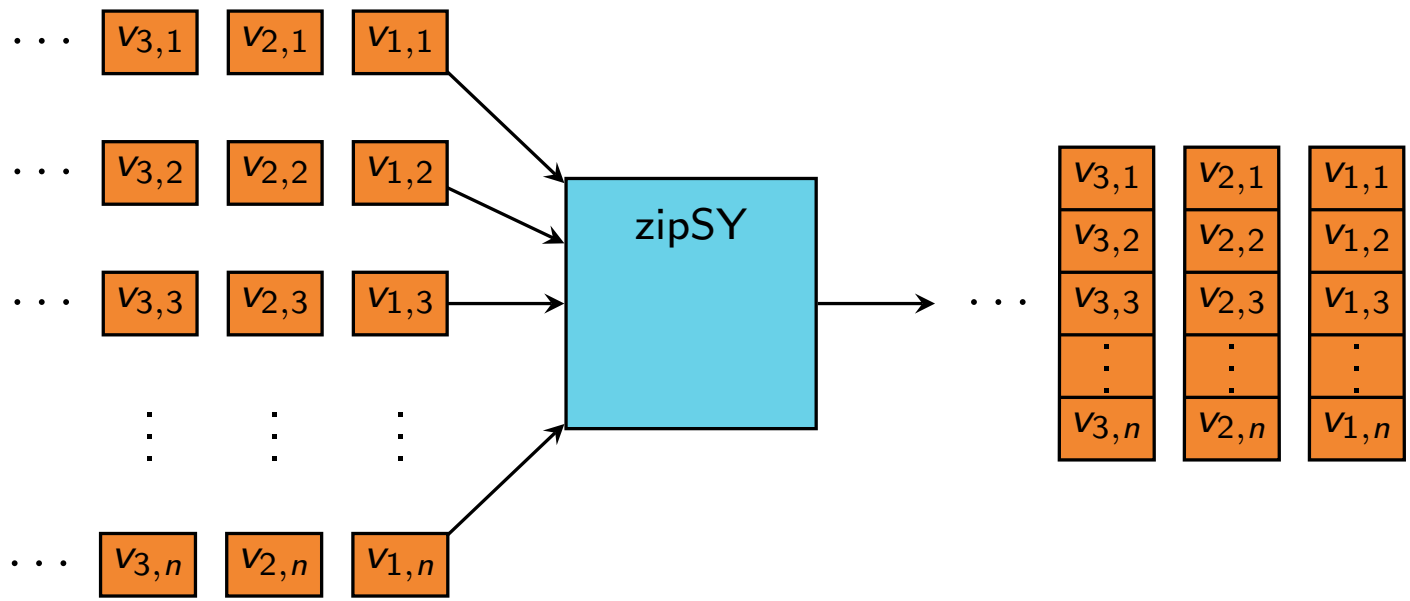
The ZipSY Process Constructor



The ZipSY Process Constructor

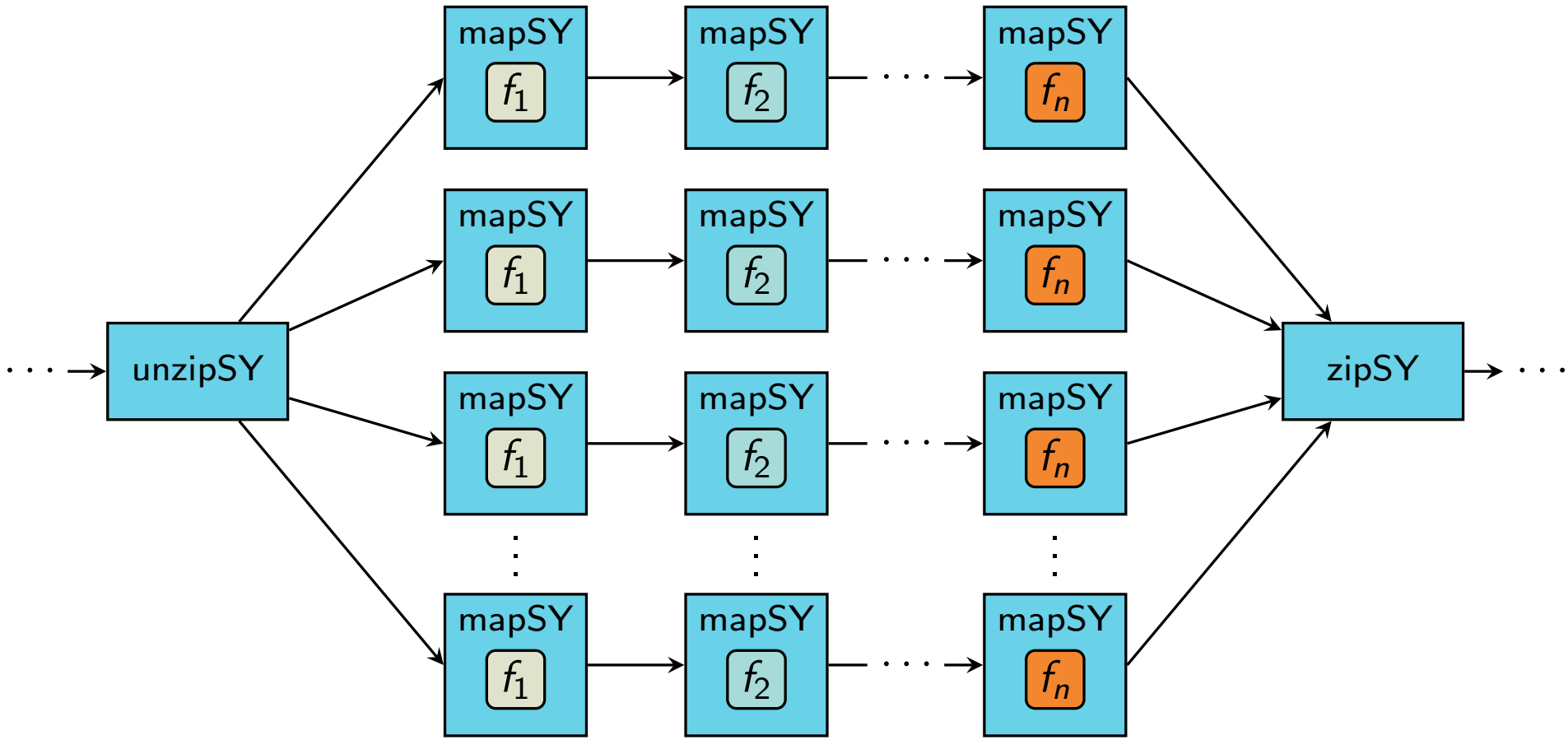


The ZipSY Process Constructor



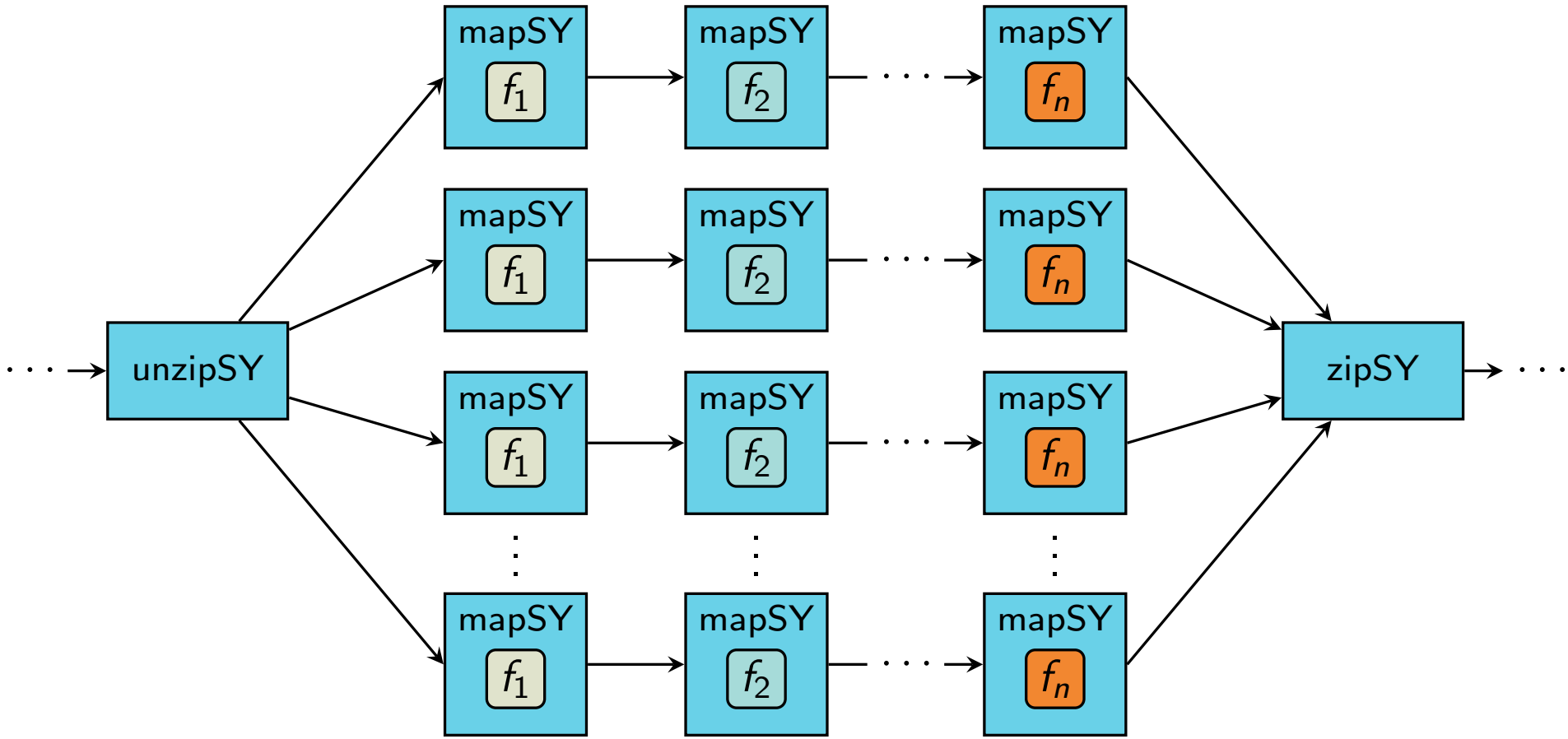
ForSyDe Models Suitable for GPGPUs?

ForSyDe Models Suitable for GPGPUs?

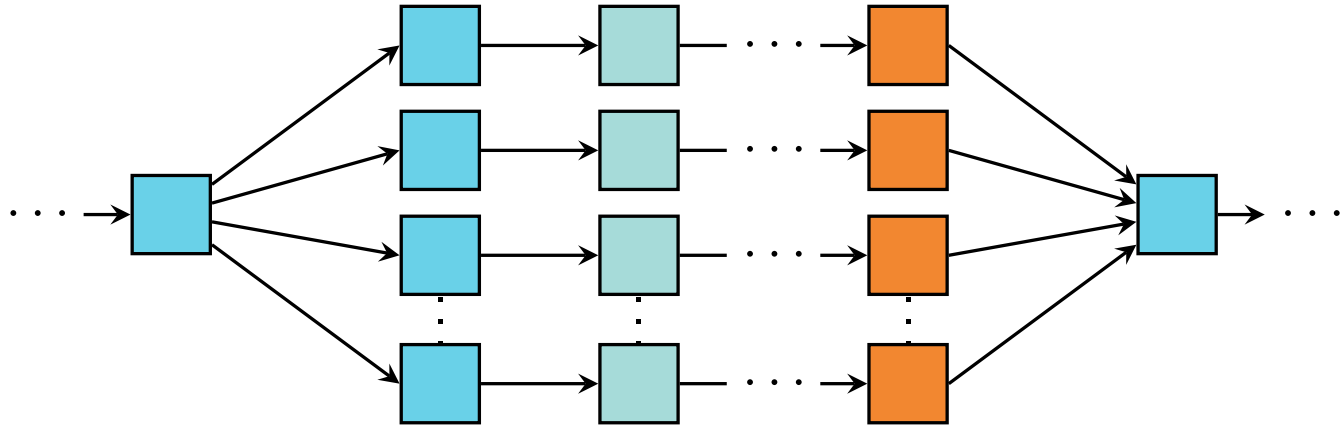


The *split-map-merge* pattern

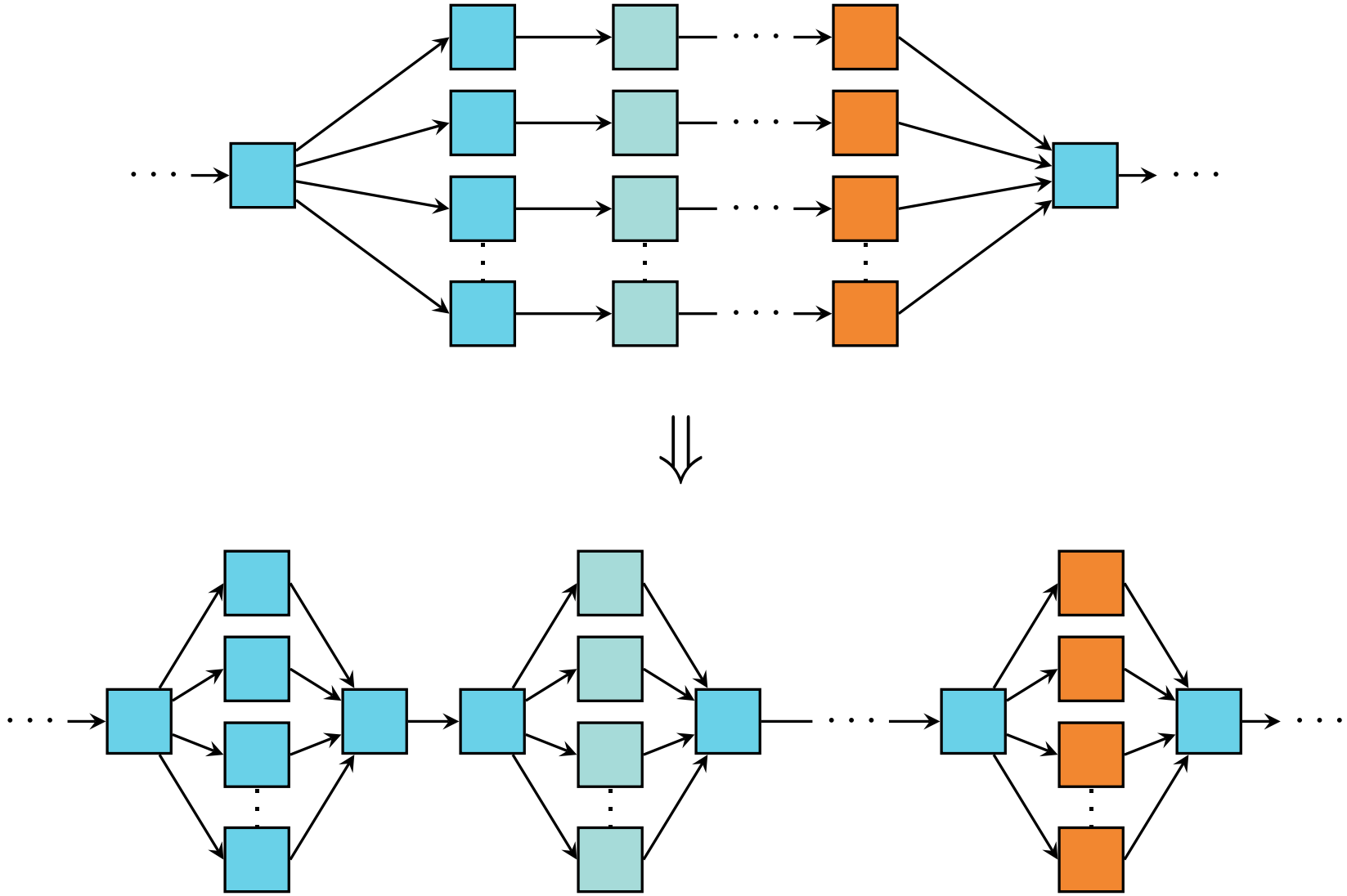
Want to Handle Only One Function



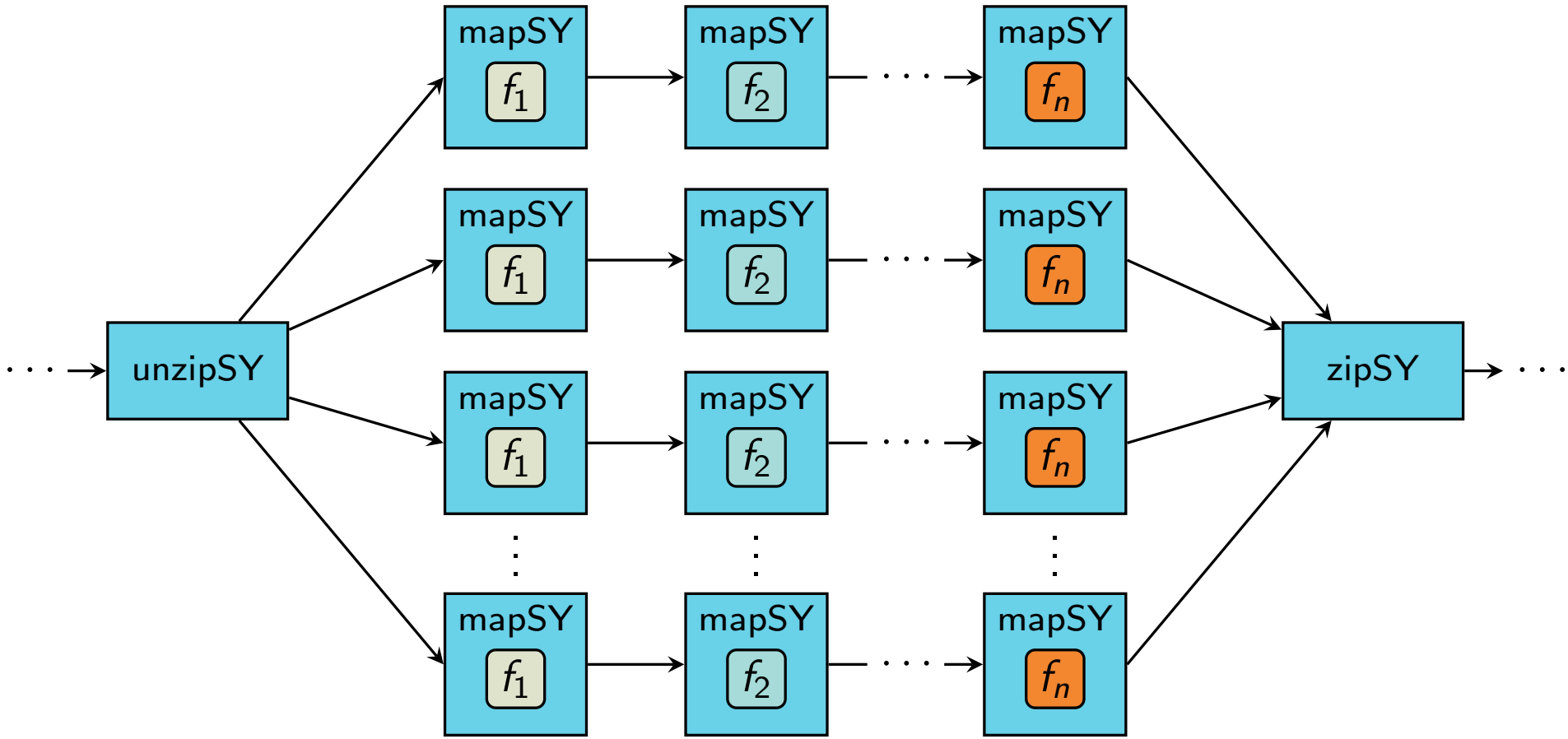
Method 1: *Section Splitting*



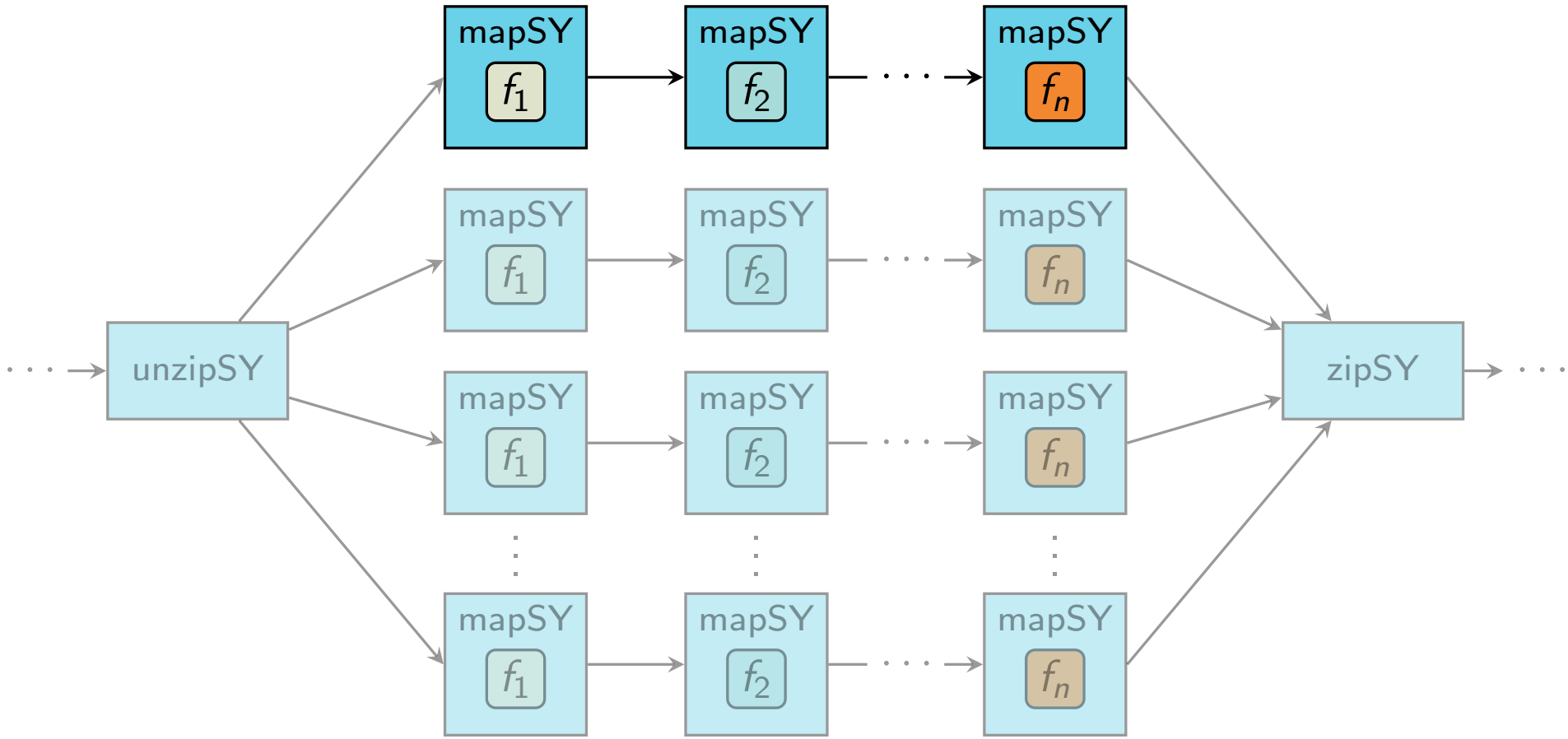
Method 1: *Section Splitting*



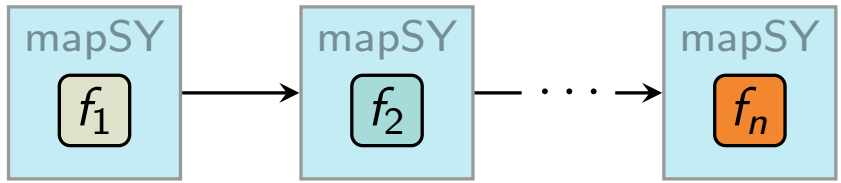
Method 2: *Process Coalescing*



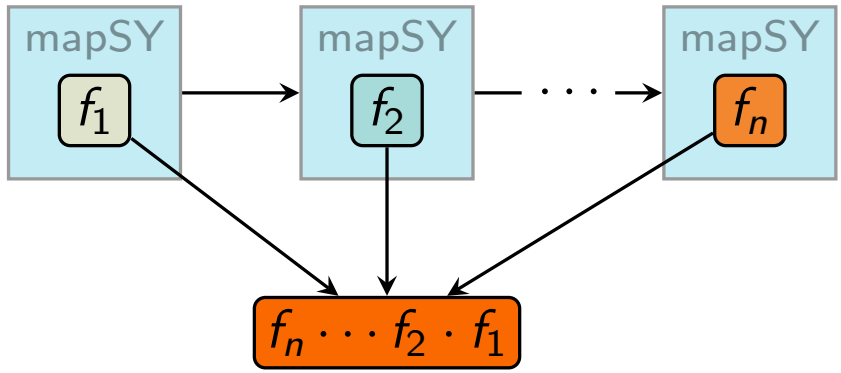
Method 2: *Process Coalescing*



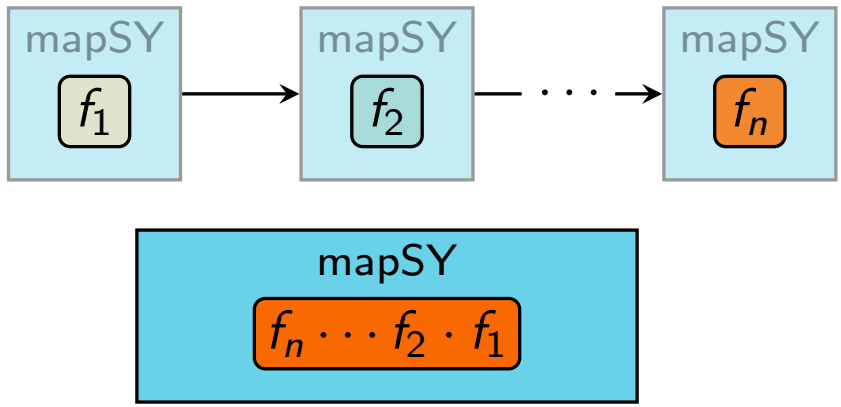
Method 2: *Process Coalescing*



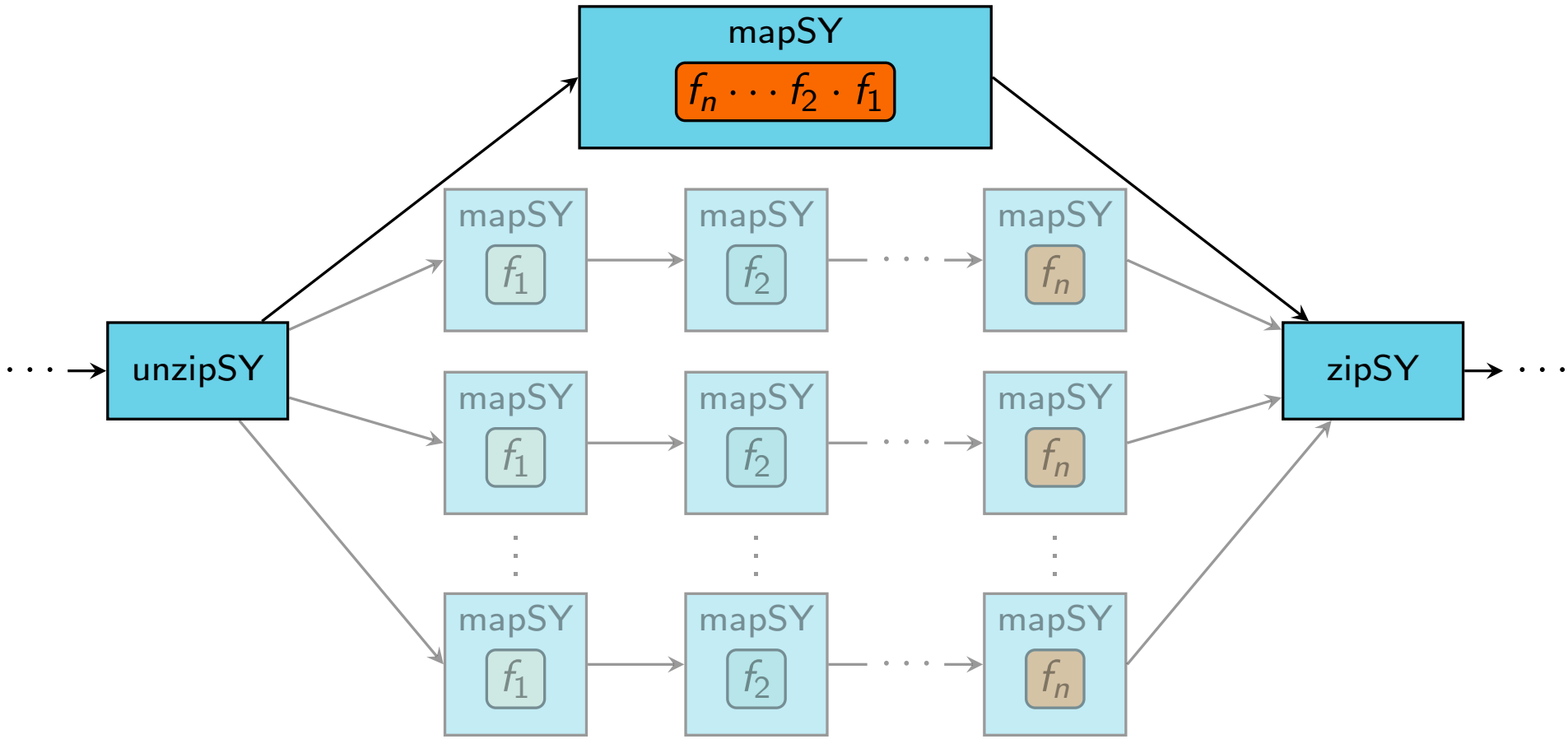
Method 2: *Process Coalescing*



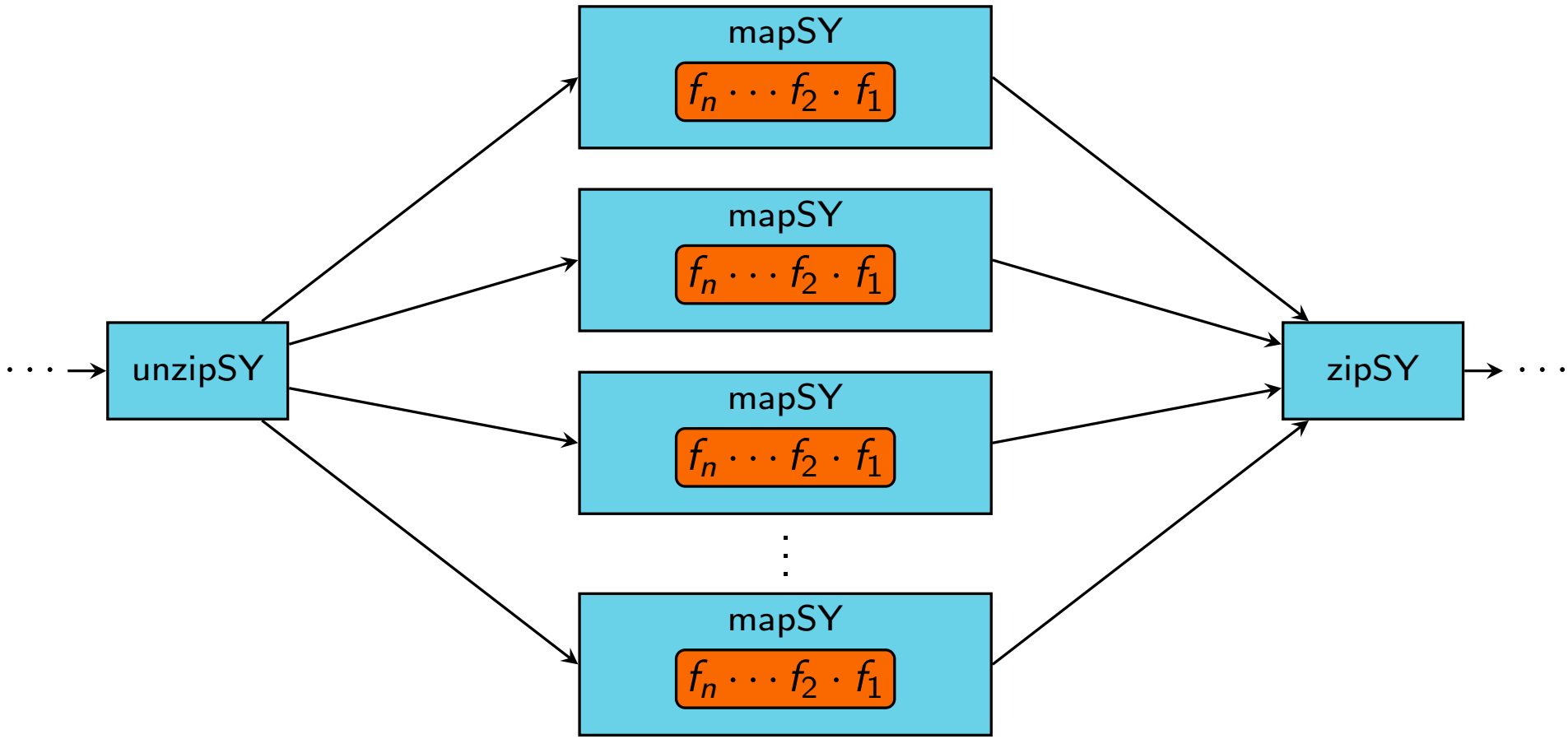
Method 2: *Process Coalescing*



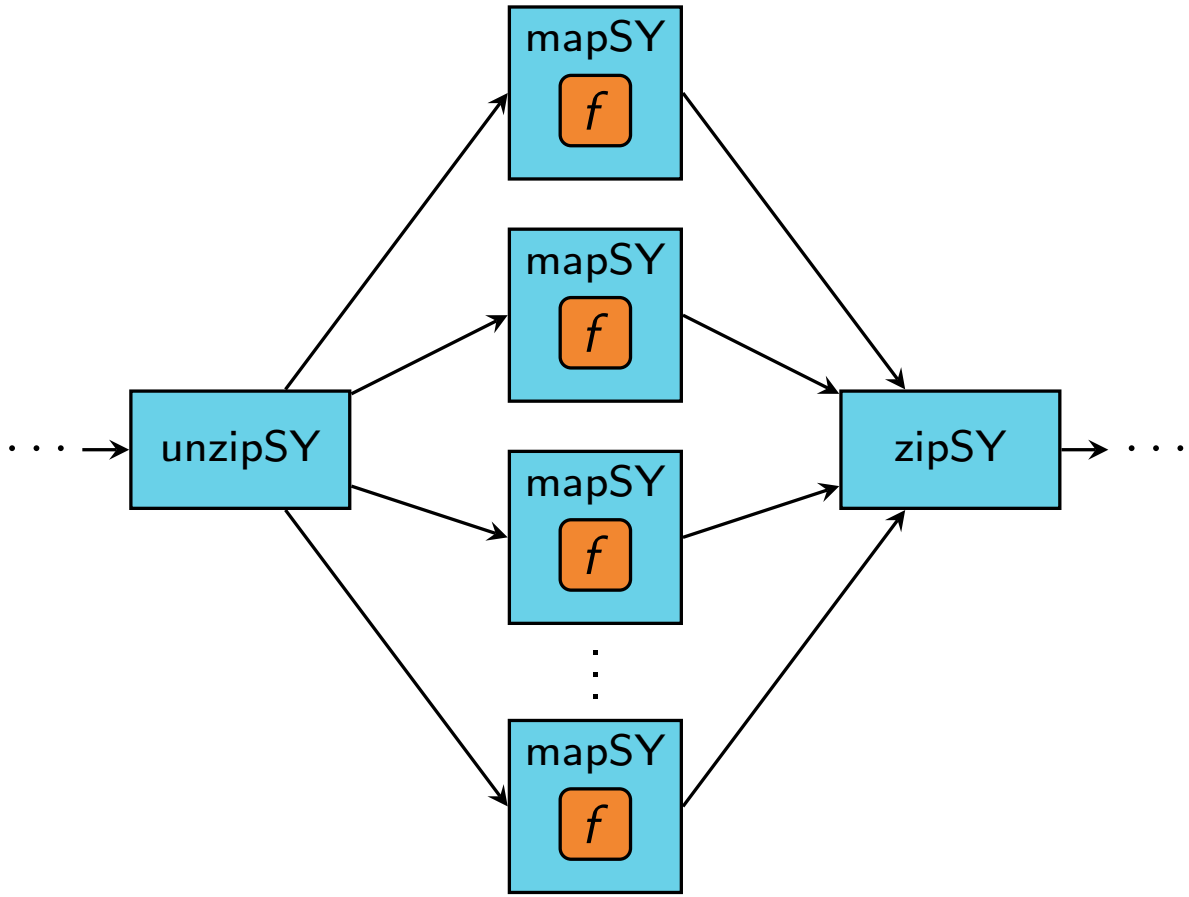
Method 2: *Process Coalescing*



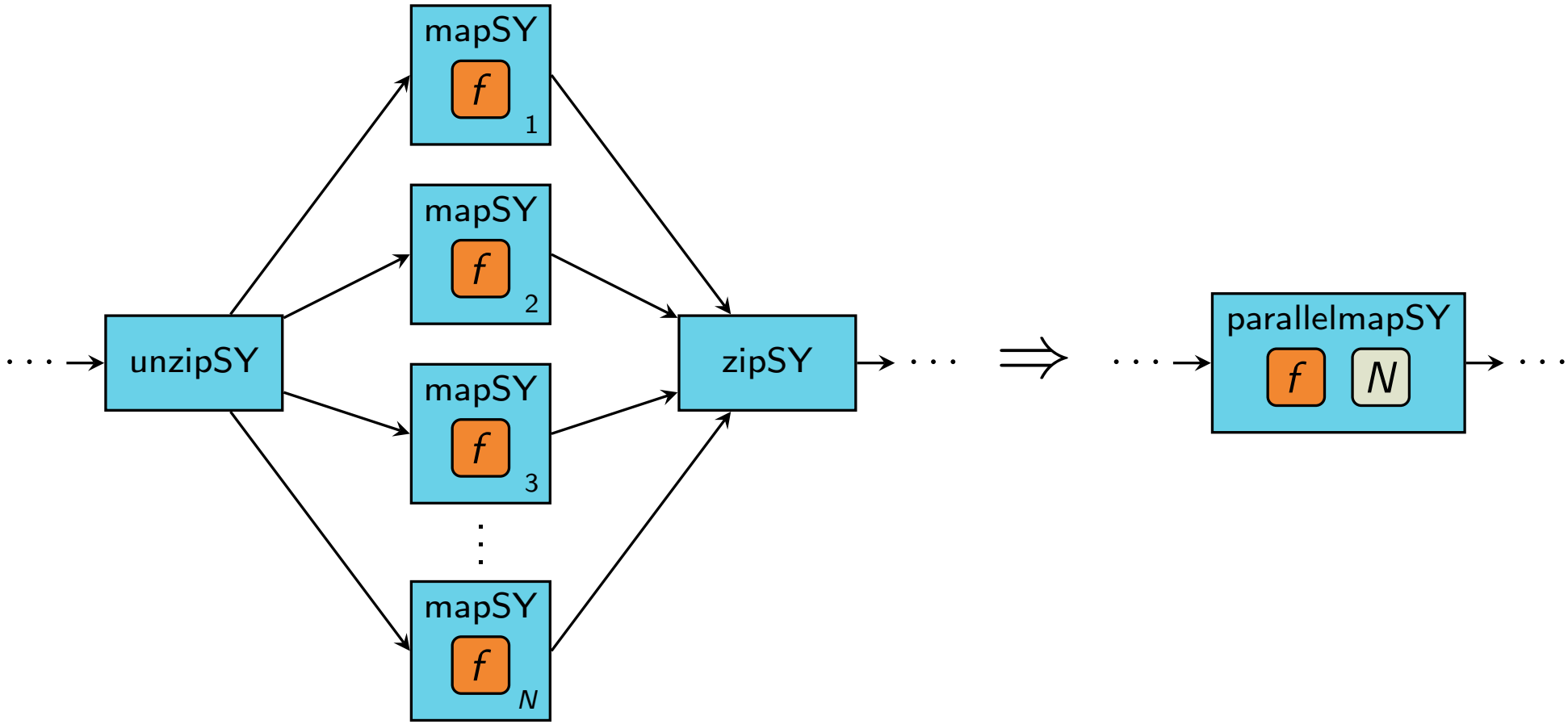
Method 2: *Process Coalescing*



Fuse Zip-Map-Unzip Structures Into *ParallelMaps*



Fuse Zip-Map-Unzip Structures Into *ParallelMaps*



Synthesis

Synthesis

- ParallelmapSY processes:

Synthesis

- ParallelmapSY processes:
  - Choose C or CUDA C implementation

Synthesis

- ParallelmapSY processes:
  - Choose C or CUDA C implementation
  - Choose to use shared memory in CUDA C implementation

Synthesis

- ParallelmapSY processes:
  - Choose C or CUDA C implementation
  - Choose to use shared memory in CUDA C implementation
- Other processes:

Synthesis

- ParallelmapSY processes:
  - Choose C or CUDA C implementation
  - Choose to use shared memory in CUDA C implementation
- Other processes:
  - Always C implementation

f2cc

f2cc

- *ForSyDe-2-CUDA C*

f2cc

- *ForSyDe-2-CUDA C*
- Proof-of-concept synthesis tool

f2cc

- *ForSyDe-2-CUDA C*
- Proof-of-concept synthesis tool
- Assumptions:

f2cc

- *ForSyDe-2-CUDA C*
- Proof-of-concept synthesis tool
- Assumptions:
  1. All functions are written in C

f2cc

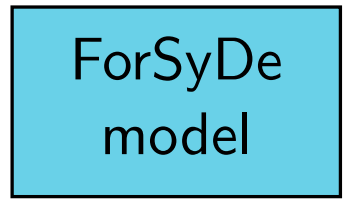
- *ForSyDe-2-CUDA C*
- Proof-of-concept synthesis tool
- Assumptions:
  1. All functions are written in C
  2. All processes are based on synchronous MoC

f2cc

- *ForSyDe-2-CUDA C*
- Proof-of-concept synthesis tool
- Assumptions:
  1. All functions are written in C
  2. All processes are based on synchronous MoC
- Design flow:

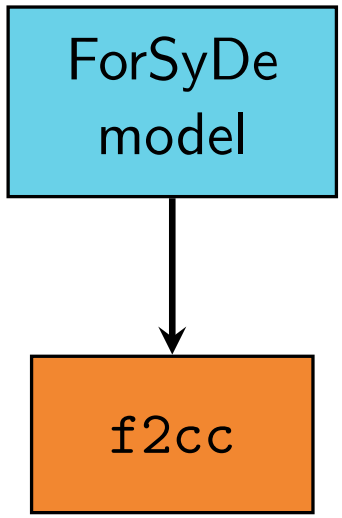
f2cc

- *ForSyDe-2-CUDA C*
- Proof-of-concept synthesis tool
- Assumptions:
  1. All functions are written in C
  2. All processes are based on synchronous MoC
- Design flow:
  1. Design ForSyDe model



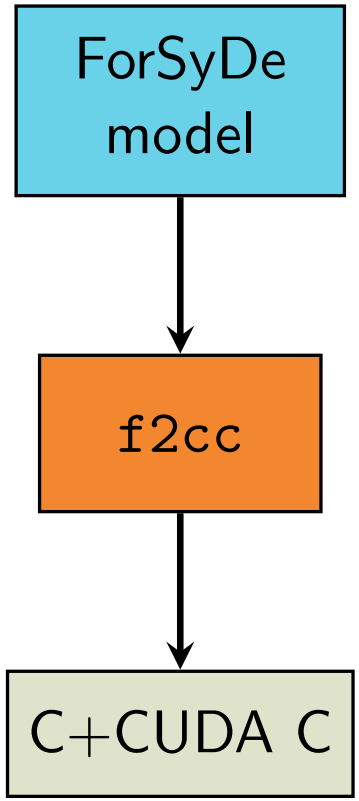
f2cc

- *ForSyDe-2-CUDA C*
- Proof-of-concept synthesis tool
- Assumptions:
  1. All functions are written in C
  2. All processes are based on synchronous MoC
- Design flow:
  1. Design ForSyDe model
  2. Run `f2cc` on ForSyDe model



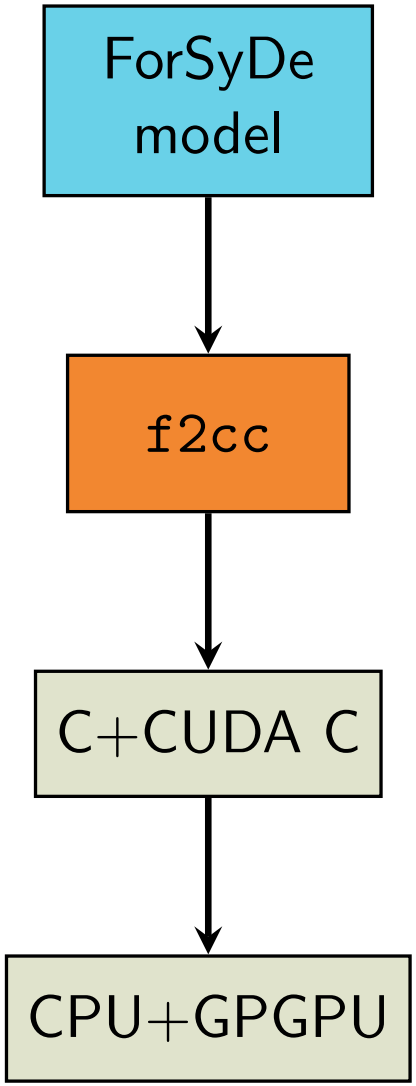
f2cc

- *ForSyDe-2-CUDA C*
- Proof-of-concept synthesis tool
- Assumptions:
  1. All functions are written in C
  2. All processes are based on synchronous MoC
- Design flow:
  1. Design ForSyDe model
  2. Run `f2cc` on ForSyDe model
  3. Get implementation in C+CUDA C



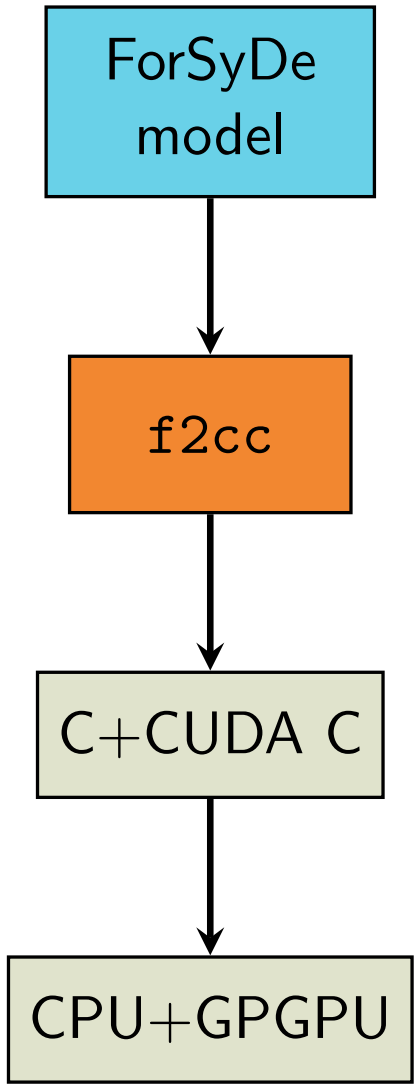
f2cc

- *ForSyDe-2-CUDA C*
- Proof-of-concept synthesis tool
- Assumptions:
  1. All functions are written in C
  2. All processes are based on synchronous MoC
- Design flow:
  1. Design ForSyDe model
  2. Run `f2cc` on ForSyDe model
  3. Get implementation in C+CUDA C
  4. Compile and execute



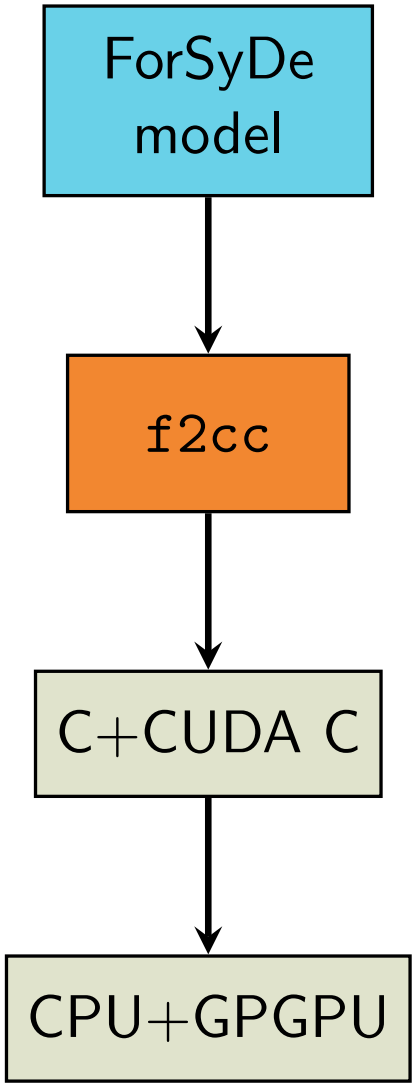
f2cc

- *ForSyDe-2-CUDA C*
- Proof-of-concept synthesis tool
- Assumptions:
  1. All functions are written in C
  2. All processes are based on synchronous MoC
- Design flow:
  1. Design ForSyDe model
  2. Run `f2cc` on ForSyDe model
  3. Get implementation in C+CUDA C
  4. Compile and execute
- Other aspects in paper but not in talk:



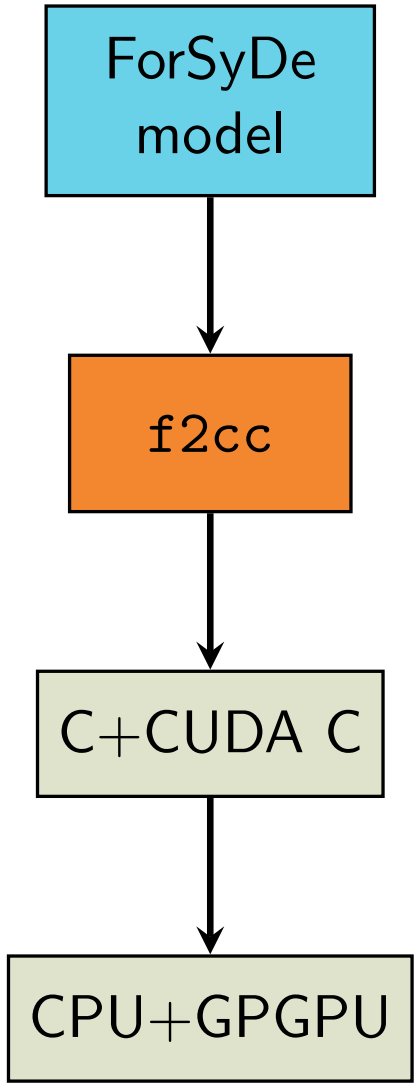
f2cc

- *ForSyDe-2-CUDA C*
- Proof-of-concept synthesis tool
- Assumptions:
  1. All functions are written in C
  2. All processes are based on synchronous MoC
- Design flow:
  1. Design ForSyDe model
  2. Run `f2cc` on ForSyDe model
  3. Get implementation in C+CUDA C
  4. Compile and execute
- Other aspects in paper but not in talk:
  - Process scheduling



f2cc

- *ForSyDe-2-CUDA C*
- Proof-of-concept synthesis tool
- Assumptions:
  1. All functions are written in C
  2. All processes are based on synchronous MoC
- Design flow:
  1. Design ForSyDe model
  2. Run `f2cc` on ForSyDe model
  3. Get implementation in C+CUDA C
  4. Compile and execute
- Other aspects in paper but not in talk:
  - Process scheduling
  - Signal management



Experiments: Setup and Expectations

Experiments: Setup and Expectations

- Setup:

Experiments: Setup and Expectations

- Setup:
  - Tested `f2cc` on two ForSyDe models

Experiments: Setup and Expectations

- Setup:
  - Tested `f2cc` on two ForSyDe models
    - Mandelbrot application

Experiments: Setup and Expectations

- Setup:
  - Tested `f2cc` on two ForSyDe models
    - Mandelbrot application
    - Industrial-scale image processing application

Experiments: Setup and Expectations

- Setup:
  - Tested `f2cc` on two ForSyDe models
    - Mandelbrot application
    - Industrial-scale image processing application
  - Executed synthesized code on Intel i7 + NVIDIA GPGPU with 96 cores

Experiments: Setup and Expectations

- Setup:
  - Tested `f2cc` on two ForSyDe models
    - Mandelbrot application
    - Industrial-scale image processing application
  - Executed synthesized code on Intel i7 + NVIDIA GPGPU with 96 cores
  - Compared execution times against hand-written C implementations

Experiments: Setup and Expectations

- Setup:
  - Tested `f2cc` on two ForSyDe models
    - Mandelbrot application
    - Industrial-scale image processing application
  - Executed synthesized code on Intel i7 + NVIDIA GPGPU with 96 cores
  - Compared execution times against hand-written C implementations
- Expected outcome:

Experiments: Setup and Expectations

- Setup:
  - Tested `f2cc` on two ForSyDe models
    - Mandelbrot application
    - Industrial-scale image processing application
  - Executed synthesized code on Intel i7 + NVIDIA GPGPU with 96 cores
  - Compared execution times against hand-written C implementations
- Expected outcome:
  - Synthesized C to perform no worse

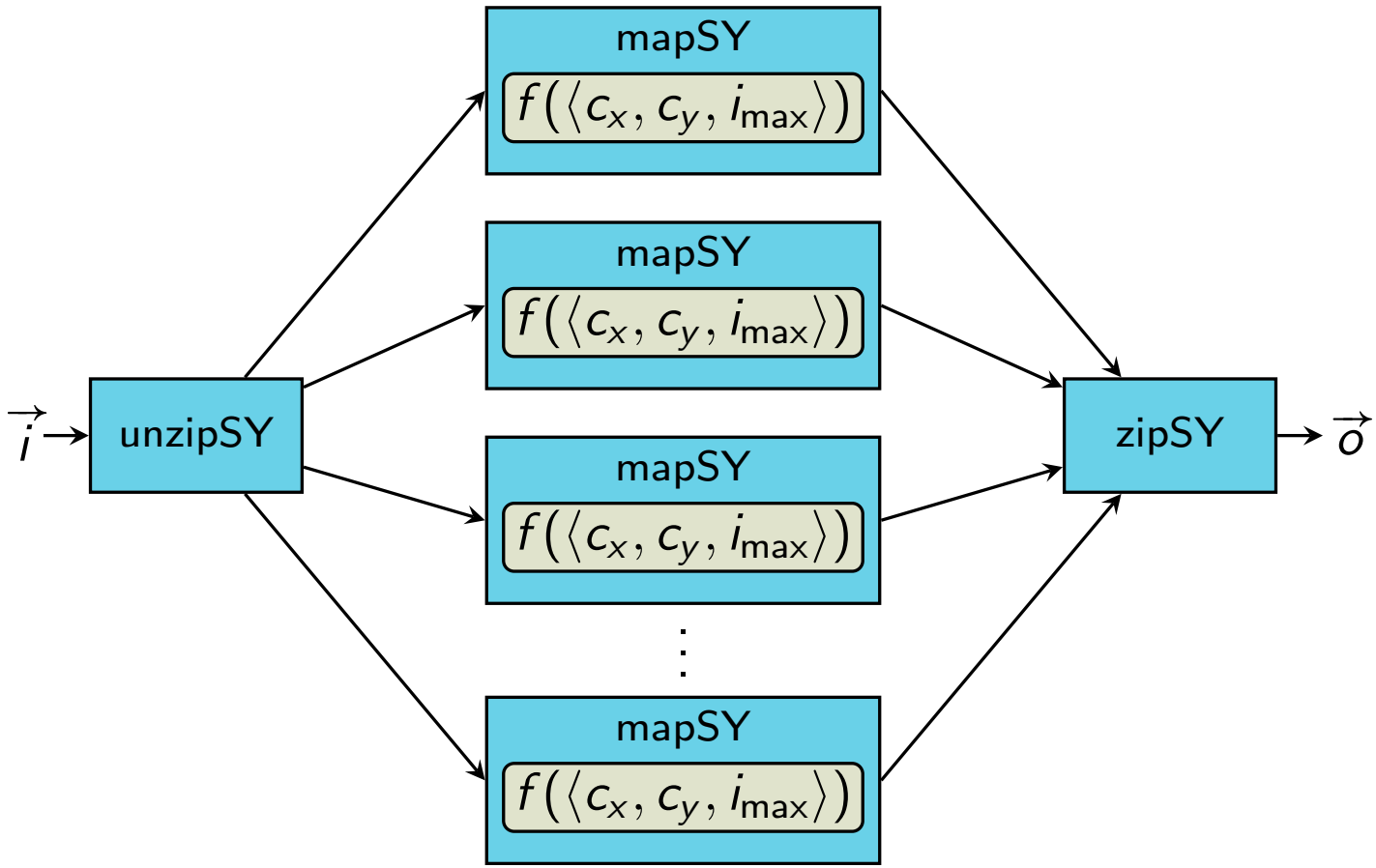
Experiments: Setup and Expectations

- Setup:
  - Tested `f2cc` on two ForSyDe models
    - Mandelbrot application
    - Industrial-scale image processing application
  - Executed synthesized code on Intel i7 + NVIDIA GPGPU with 96 cores
  - Compared execution times against hand-written C implementations
- Expected outcome:
  - Synthesized C to perform no worse
  - Synthesized CUDA C to perform better

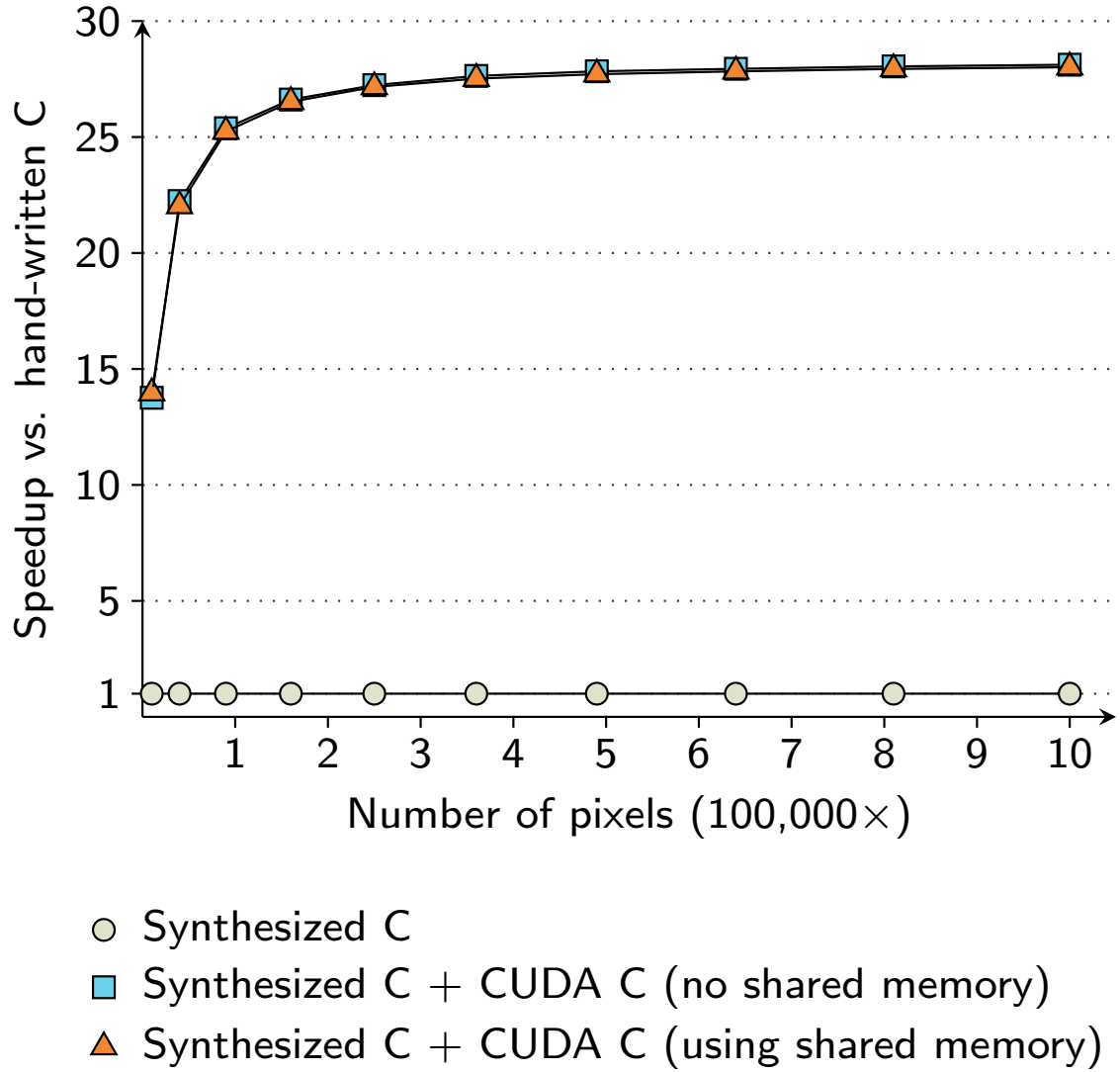
Experiments: Setup and Expectations

- Setup:
  - Tested `f2cc` on two ForSyDe models
    - Mandelbrot application
    - Industrial-scale image processing application
  - Executed synthesized code on Intel i7 + NVIDIA GPGPU with 96 cores
  - Compared execution times against hand-written C implementations
- Expected outcome:
  - Synthesized C to perform no worse
  - Synthesized CUDA C to perform better
    - $\backsim$ 30$\times$ speedup is good

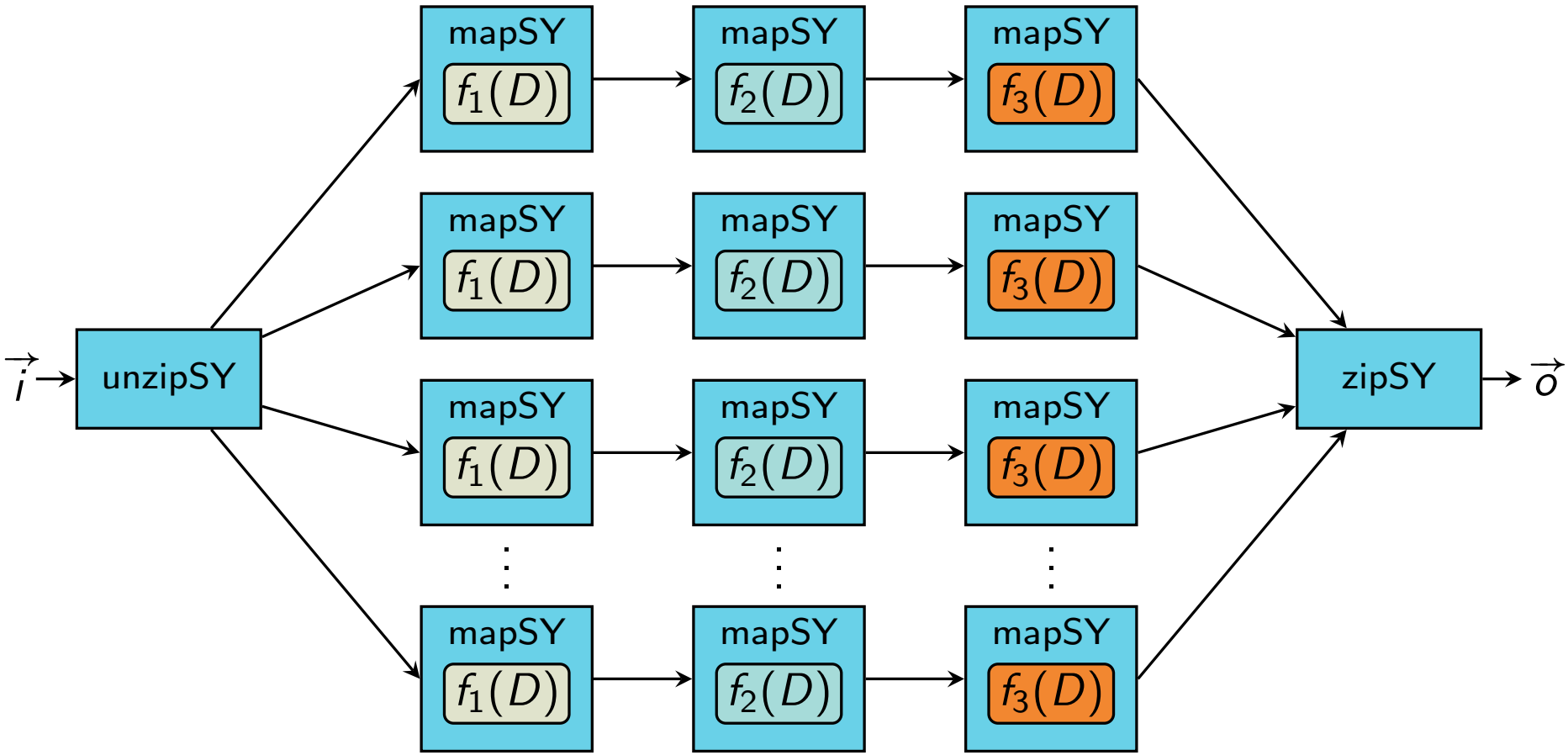
ForSyDe Model of Mandelbrot Application



Achieve Expected Outcome for Mandelbrot Application

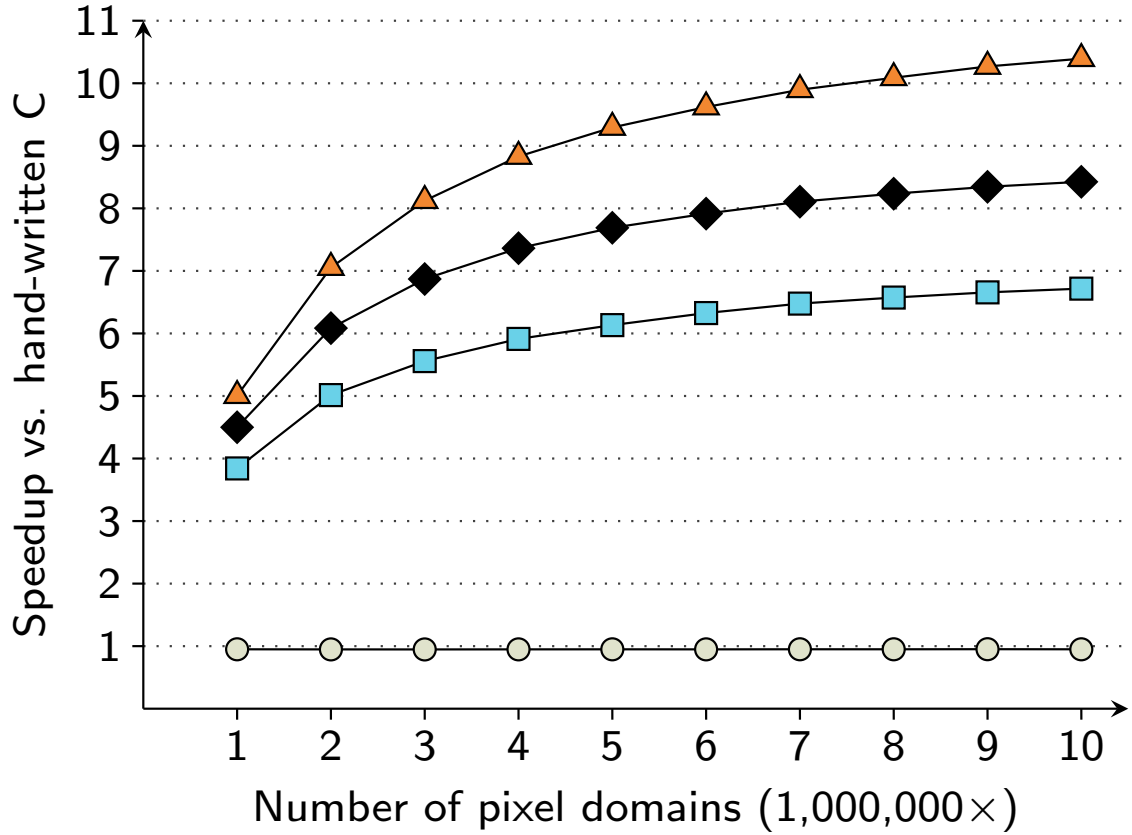


ForSyDe Model of Image Processing Application



$$D = \langle p_1, p_2, p_3, p_4, p_5, p_6, p_7 \rangle$$

Achieve Expected Outcome for Image Processing Application



- Synthesized C
- Synthesized C + CUDA C (section splitting, no shared memory)
- Synthesized C + CUDA C (process coalescing, no shared memory)
- Synthesized C + CUDA C (process coalescing, with shared memory)

Discussion of Experiment Results

Discussion of Experiment Results

- Mandelbrot more compute-intense than image proc. app.

Discussion of Experiment Results

- Mandelbrot more compute-intense than image proc. app.
  $\Rightarrow$ more offset of GPGPU overhead for Mandelbrot

Discussion of Experiment Results

- Mandelbrot more compute-intense than image proc. app.
  $\Rightarrow$ more offset of GPGPU overhead for Mandelbrot
  $\Rightarrow$ more speedup for Mandelbrot

Discussion of Experiment Results

- Mandelbrot more compute-intense than image proc. app.
  $\Rightarrow$ more offset of GPGPU overhead for Mandelbrot
  $\Rightarrow$ more speedup for Mandelbrot
- Section splitting leads to excess memory copying

Discussion of Experiment Results

- Mandelbrot more compute-intense than image proc. app.
  $\Rightarrow$ more offset of GPGPU overhead for Mandelbrot
  $\Rightarrow$ more speedup for Mandelbrot
- Section splitting leads to excess memory copying
  $\Rightarrow$ higher GPGPU overhead for section splitting

Discussion of Experiment Results

- Mandelbrot more compute-intense than image proc. app.
  $\Rightarrow$ more offset of GPGPU overhead for Mandelbrot
  $\Rightarrow$ more speedup for Mandelbrot
- Section splitting leads to excess memory copying
  $\Rightarrow$ higher GPGPU overhead for section splitting
  $\Rightarrow$ more speedup when using process coalescing

Discussion of Experiment Results

- Mandelbrot more compute-intense than image proc. app.
  $\Rightarrow$ more offset of GPGPU overhead for Mandelbrot
  $\Rightarrow$ more speedup for Mandelbrot
- Section splitting leads to excess memory copying
  $\Rightarrow$ higher GPGPU overhead for section splitting
  $\Rightarrow$ more speedup when using process coalescing
- More shared memory per thread for image proc. app. than Mandelbrot

Discussion of Experiment Results

- Mandelbrot more compute-intense than image proc. app.
  $\Rightarrow$ more offset of GPGPU overhead for Mandelbrot
  $\Rightarrow$ more speedup for Mandelbrot
- Section splitting leads to excess memory copying
  $\Rightarrow$ higher GPGPU overhead for section splitting
  $\Rightarrow$ more speedup when using process coalescing
- More shared memory per thread for image proc. app. than Mandelbrot
  $\Rightarrow$ over-use of shared memory for image proc. app.

Discussion of Experiment Results

- Mandelbrot more compute-intense than image proc. app.
  $\Rightarrow$ more offset of GPGPU overhead for Mandelbrot
  $\Rightarrow$ more speedup for Mandelbrot
- Section splitting leads to excess memory copying
  $\Rightarrow$ higher GPGPU overhead for section splitting
  $\Rightarrow$ more speedup when using process coalescing
- More shared memory per thread for image proc. app. than Mandelbrot
  $\Rightarrow$ over-use of shared memory for image proc. app.
  $\Rightarrow$ more speedup when not using shared memory

Current Status & Future Work

Current Status & Future Work

- Proof-of-concept prototype for split-map-merge pattern

Current Status & Future Work

- Proof-of-concept prototype for split-map-merge pattern
  - Extend support for additional process constructors

Current Status & Future Work

- Proof-of-concept prototype for split-map-merge pattern
  - Extend support for additional process constructors
  - Extend support for additional patterns

Current Status & Future Work

- Proof-of-concept prototype for split-map-merge pattern
  - Extend support for additional process constructors
  - Extend support for additional patterns
  - Extend support for additional MoCs

Current Status & Future Work

- Proof-of-concept prototype for split-map-merge pattern
  - Extend support for additional process constructors
  - Extend support for additional patterns
  - Extend support for additional MoCs
- Greedy evaluation – always implement on GPGPU

Current Status & Future Work

- Proof-of-concept prototype for split-map-merge pattern
  - Extend support for additional process constructors
  - Extend support for additional patterns
  - Extend support for additional MoCs
- Greedy evaluation – always implement on GPGPU
  - Experimental cost analysis

Current Status & Future Work

- Proof-of-concept prototype for split-map-merge pattern
  - Extend support for additional process constructors
  - Extend support for additional patterns
  - Extend support for additional MoCs
- Greedy evaluation – always implement on GPGPU
  - Experimental cost analysis
  - Look into DSE (design space exploration)

Current Status & Future Work

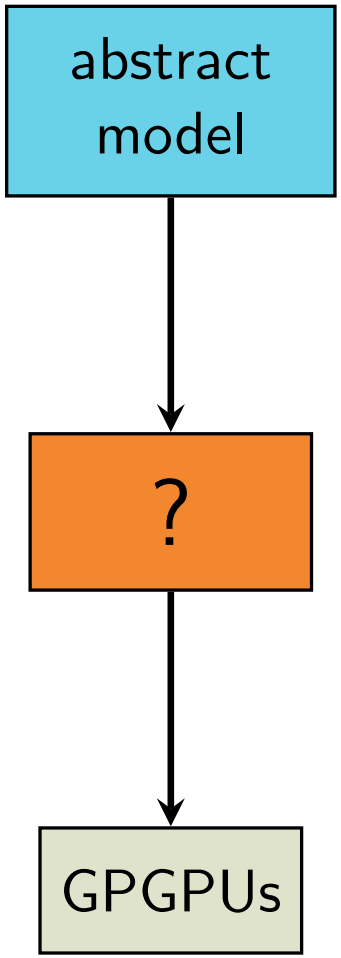
- Proof-of-concept prototype for split-map-merge pattern
  - Extend support for additional process constructors
  - Extend support for additional patterns
  - Extend support for additional MoCs
- Greedy evaluation – always implement on GPGPU
  - Experimental cost analysis
  - Look into DSE (design space exploration)
- Eager memory-copying scheme

Current Status & Future Work

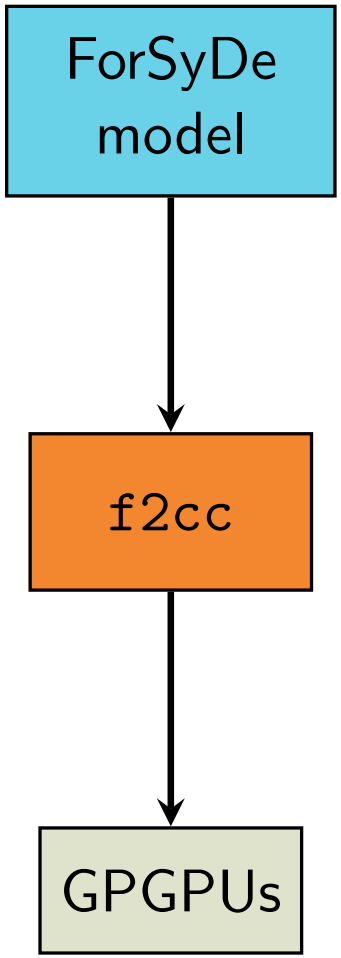
- Proof-of-concept prototype for split-map-merge pattern
  - Extend support for additional process constructors
  - Extend support for additional patterns
  - Extend support for additional MoCs
- Greedy evaluation – always implement on GPGPU
  - Experimental cost analysis
  - Look into DSE (design space exploration)
- Eager memory-copying scheme
  - Reduce overhead through lazy copying

Summary

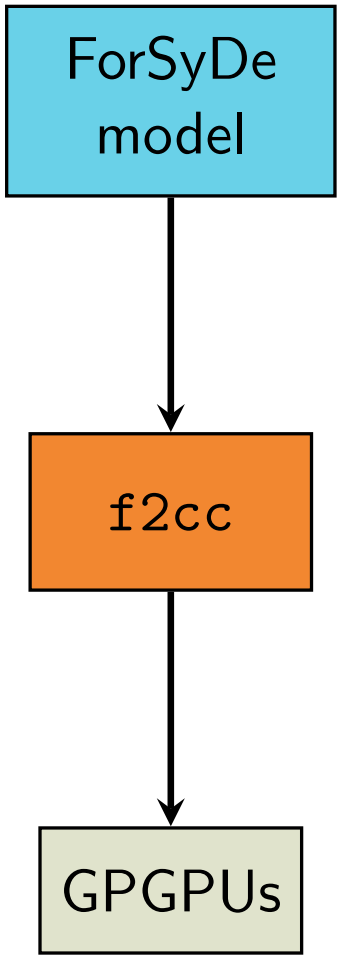
Summary



Summary

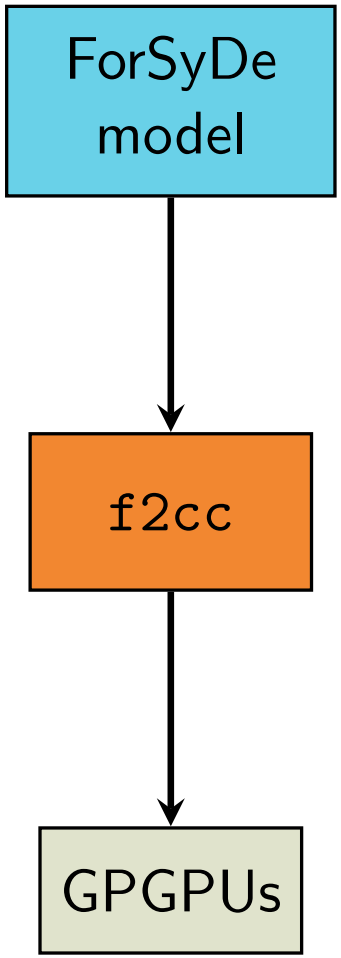


Summary



- Get high-performance implementation for supported design pattern

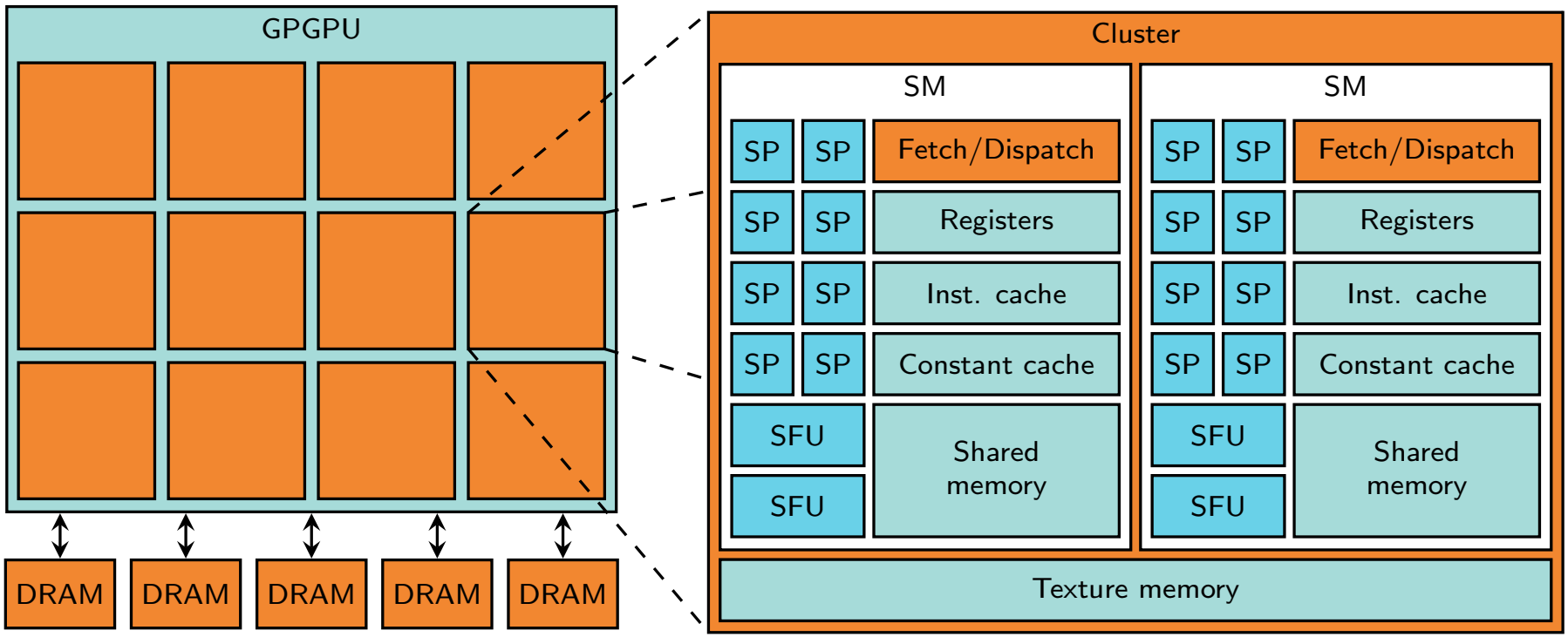
Summary



- Get high-performance implementation for supported design pattern

`f2cc` available at:
`https://forsyde.ict.kth.se/trac/wiki/ForSyDe/f2cc`

NVIDIA's GPGPU Architecture



```
<graphml>
  <graph id="test" edgedefault="directed">
    ...
    <node id="unzip"> ... </node>
    <node id="zip"> ... </node>
    <node id="map1">
      <data key="process_type">mapSY</data>
      <data key="procfun_arg">
        int f1(int x) { return x + 1; }
      </data>
      <port name="in" /><port name="out" />
    </node>
    <node id="map6">
      <data key="process_type">mapSY</data>
      <data key="procfun_arg">
        int f2(int x) { return x * 2; }
      </data>
      <port name="in" /><port name="out" />
    </node>
    <edge source="unzip" sourceport="out1" target="map1"
          targetport="in" />
    <edge source="map1" sourceport="out" target="map4"
          targetport="in" />
    ...
  </graph>
</graphml>
```

Function Produced From Process Coalescing

```
__device__
int f12(int x) {
    int res_f1 = f1(x);
    int res_f2 = f2(res_f1);
    return res_f2;
}
```

Kernel Function Produced (Without Shared Memory)

```
__global__
void f12_kernel(
  const int* input,
  int* output,
  int offset)
{
  unsigned int global_index =
    (blockIdx.x * blockDim.x + threadIdx.x) + offset;

  if (global_index < 3) {
    int input_index = global_index * 1;


    output[global_index] = f12(input[input_index]);
  }
}
```

Kernel Function Produced (With Shared Memory)

```
__global__
void f12_kernel(
  const int* input,
  int* output,
  int offset)
{
  unsigned int global_index =
    (blockIdx.x * blockDim.x + threadIdx.x) + offset;
  extern __shared__ int input_cached[];
  if (global_index < 3) {
    int input_index = threadIdx.x * 1;
    int gi_index = global_index * 1;
    input_cached[input_index + 0] = input[gi_index + 0];
    output[global_index] = f12(input_cached[input_index]);
  }
}
```

Produced Invoker Function (1 of 3)

```
void f12_invoker(const int* input, int* output) {
  int* device_input;
  int* device_output;

  struct cudaDeviceProp prop;
  cudaGetDeviceProperties(&prop, 0);

  int tlimit = prop.maxThreadsPerBlock *
               prop.multiProcessorCount;

  cudaMalloc((void**) &device_input, 3 * sizeof(int));
  cudaMalloc((void**) &device_output, 3 * sizeof(int));
  cudaMemcpy((void*) device_input,
             (void*) input,
             3 * sizeof(int),
             cudaMemcpyHostToDevice);
```

Produced Invoker Function (2 of 3)

```
if (prop.kernelExecTimeoutEnabled) {
  int num_t_left = 3;
  int offset = 0;
  while (num_t_left > 0) {
    int num_t_exec =
      num_t_left < tlimit ? num_t_left : tlimit;
    KernelConfig c = calculateBestKernelConfig(...);
    f12_kernel<<<c.grid, c.threadBlock, c.sharedMemory>>>
      (device_input, device_output, offset);
    int num_t_exed = c.grid.x * c.threadBlock.x;
    num_t_left -= num_t_exed;
    offset += num_t_exed;
  }
}
else {
  KernelConfig c = calculateBestKernelConfig(...);
  f12_kernel<<<c.grid, c.threadBlock, c.sharedMemory>>>
    (device_input, device_output, 0);
}
```

Produced Invoker Function (3 of 3)

```
  cudaMemcpy((void*) output,
             (void*) device_output,
             3 * sizeof(int),
             cudaMemcpyDeviceToHost);
  cudaFree((void*) device_input);
  cudaFree((void*) device_output);
}
```