SKIR: Just-in-Time Compilation for Parallelism with LLVM

Jeff Fifield
University of Colorado

2011 LLVM Developers Meeting
November 18, 2011

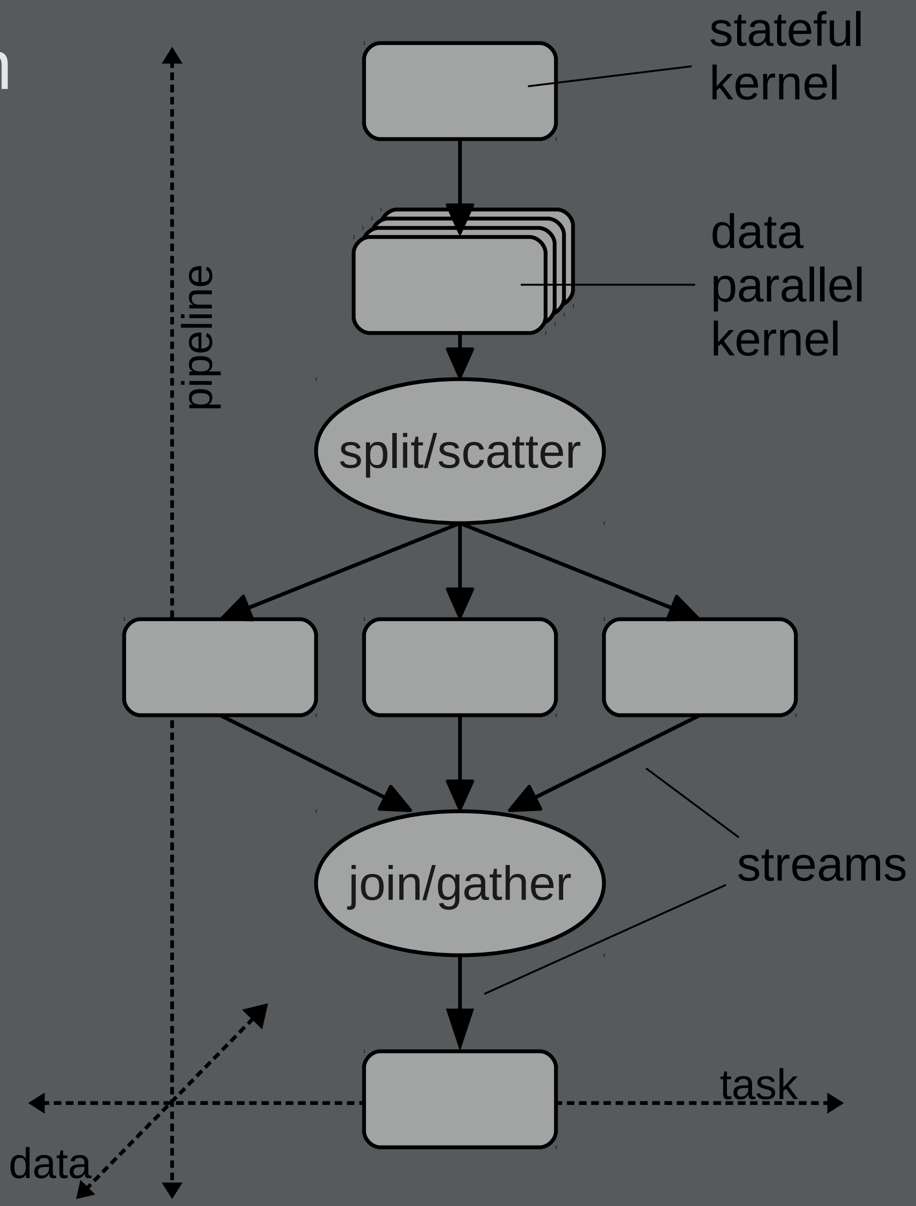
The SKIR Project is:

- Stream and Kernel Intermediate Representation (SKIR)
  - SKIR = LLVM + Streams/Kernels
- Stream language front-end compilers
- SKIR code optimizations for stream parallelism
- Dynamic scheduler for shared memory x86
- LLVM JIT back-end for CPU
- LLVM to OpenCL back-end for GPU

What is Stream Parallelism?

- Regular data-centric computation
- Independent processing stages with explicit communication
- Pipeline, Data, and Task Parallelism
- Examples:

  - Digital Signal Processing
  - Encoding/Decoding
    - Compression, Cryptography
    - Video, Audio
  - Network Processing
  - Real-time “Big Data” services



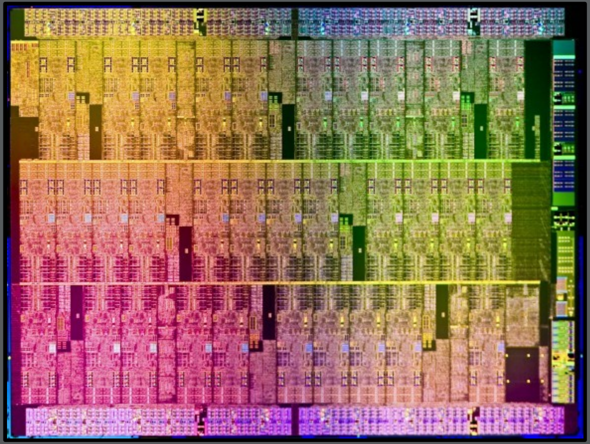
Formal Models of Stream Parallelism

- **Kahn Process Networks (KPN)** [1]
  - computation as a graph of independent processes
  - communication over unidirectional FIFO channels
  - block on read to empty channel, never block on write
  - deterministic kernels ⇒ deterministic network
- **Synchronous Data Flow Networks (SDF)** [2]
  - restriction of KPN where kernel have fixed I/O rates
  - allows better compiler analysis
  - allows static scheduling techniques

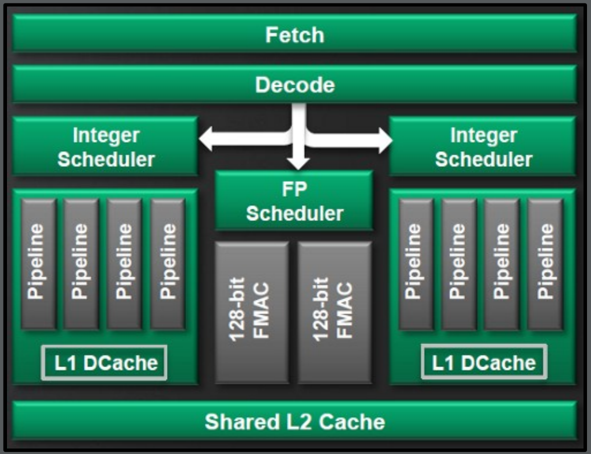
[1] G. Kahn, *The semantics of a simple language for parallel programming*, Information Processing (1974), 471–475.
[2] E. A. Lee and D. G. Messerschmitt, *Static scheduling of synchronous data flow programs for digital signal processing*, IEEE Transactions on Computing 36 (1987), no. 1, 24–35.

Why Stream Parallelism?

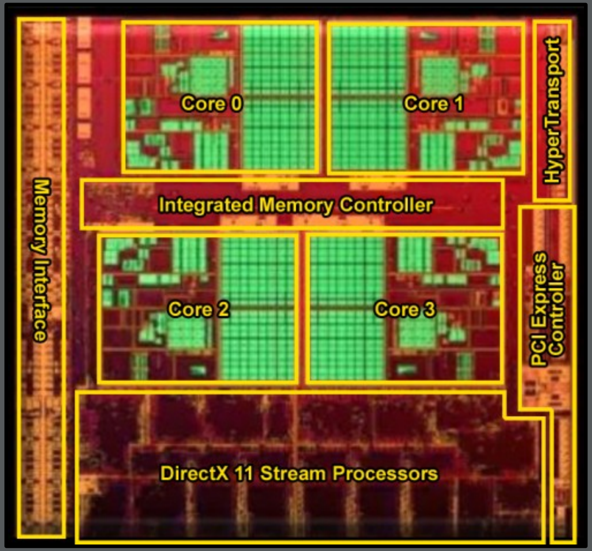
It can target increasingly diverse parallel hardware



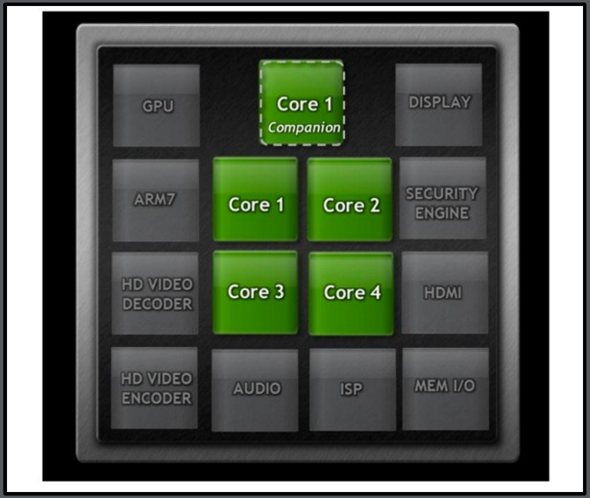
Intel Knights Corner: 50 Cores



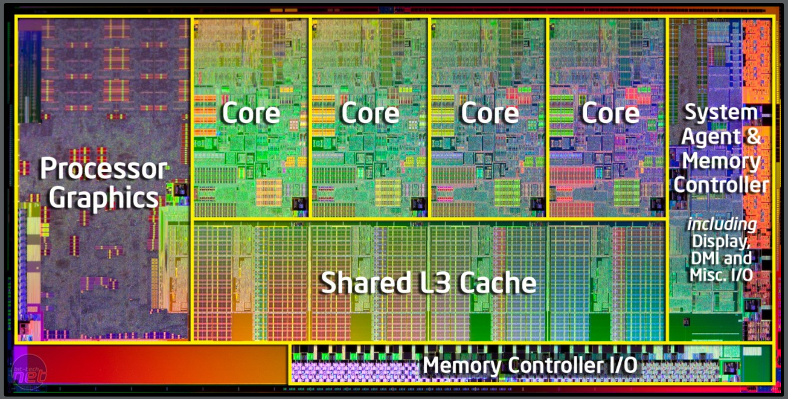
AMD Bulldozer: Shared FPU



AMD Llano: On Die GPU



NVIDIA Tegra 3: Asymmetric Multicore



Intel Sandy Bridge: Shared L3 between Gfx, CPU

Why Stream Parallelism?

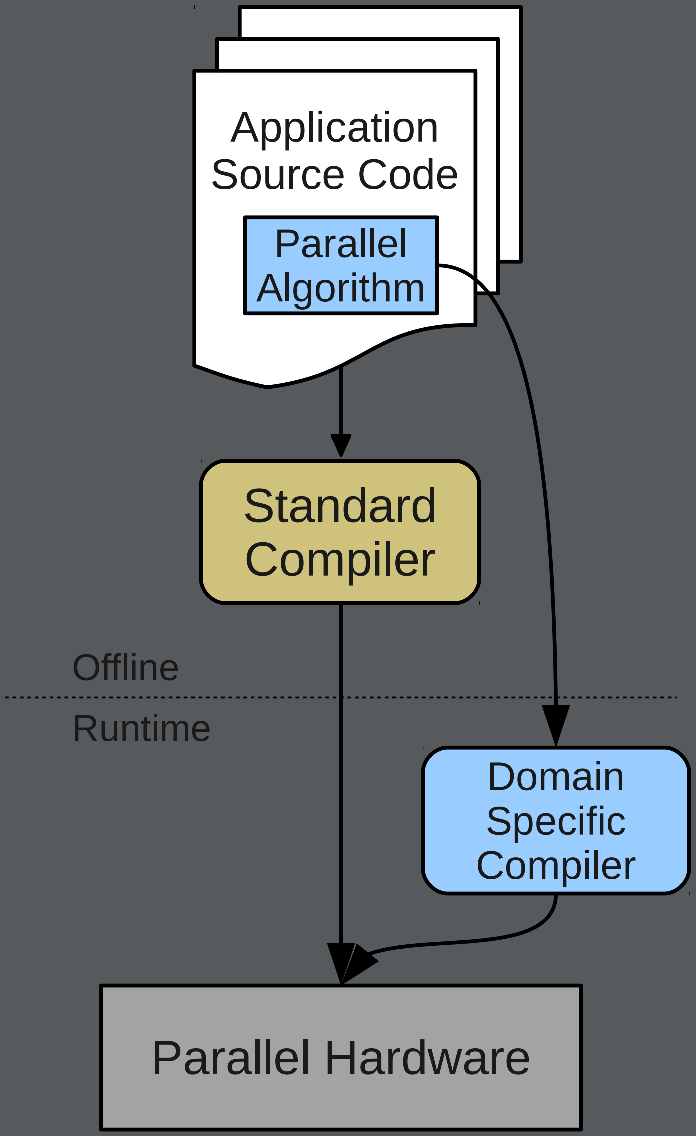
It Supports a variety of data centric applications

- Audio processing
- Image processing
- Compression
- Encryption
- Data Mining
- Software Radio
- 2-D and 3-D Graphics
- Physics Simulations
- Financial Applications
- Network Processing
- Computational Bio
- Game Physics
- Twitter Parsing
- Marmot Detection
- And many more...

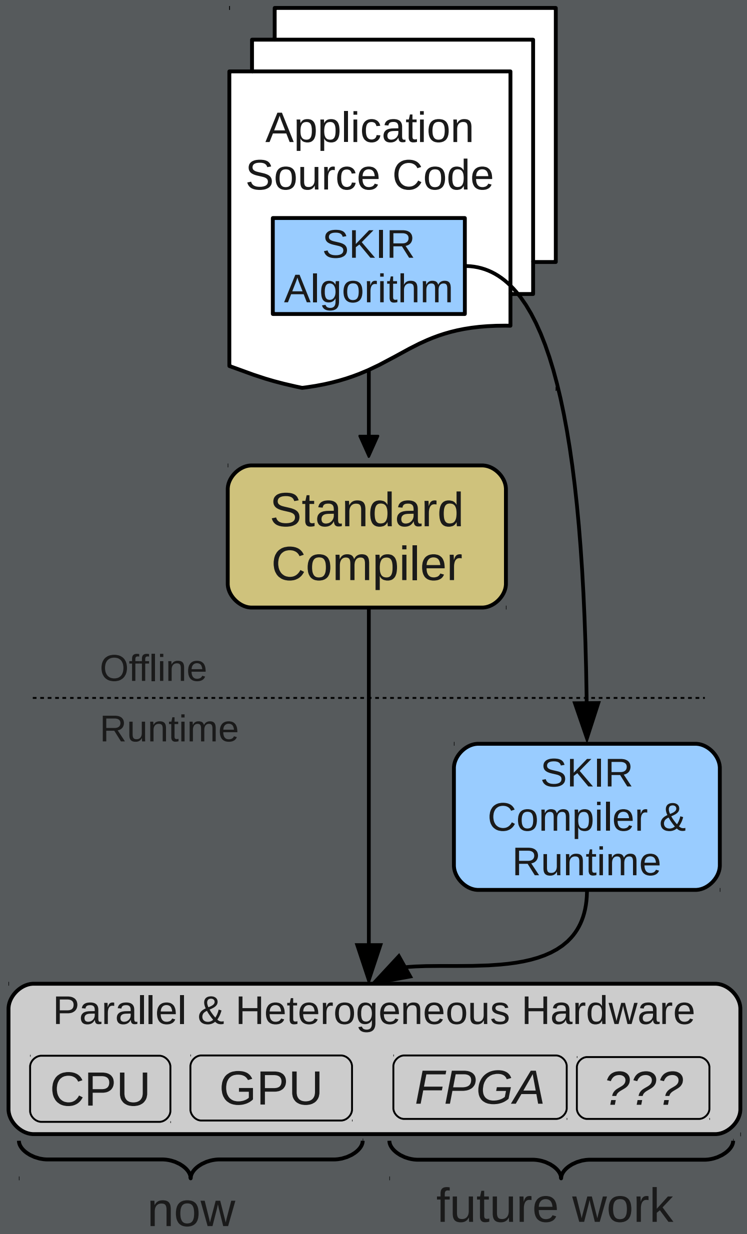
Why SKIR?

Embedded domain specific parallelism is useful

- **Embed domain specific knowledge**
  - into the language, compiler, or runtime system
- **Programming model tailored to your problem**
  - higher level of abstraction ⇒ higher productivity
  - restricted prog. model ⇒ higher performance
- **Your favorite language, with better parallelism**
- Examples:
  - PLINQ – optimizable embedded query language
  - ArBB – vector computation on parallel arrays
  - CUDA – memory and execution models for GPUs
  - SKIR – language independent stream parallelism



SKIR: Overview



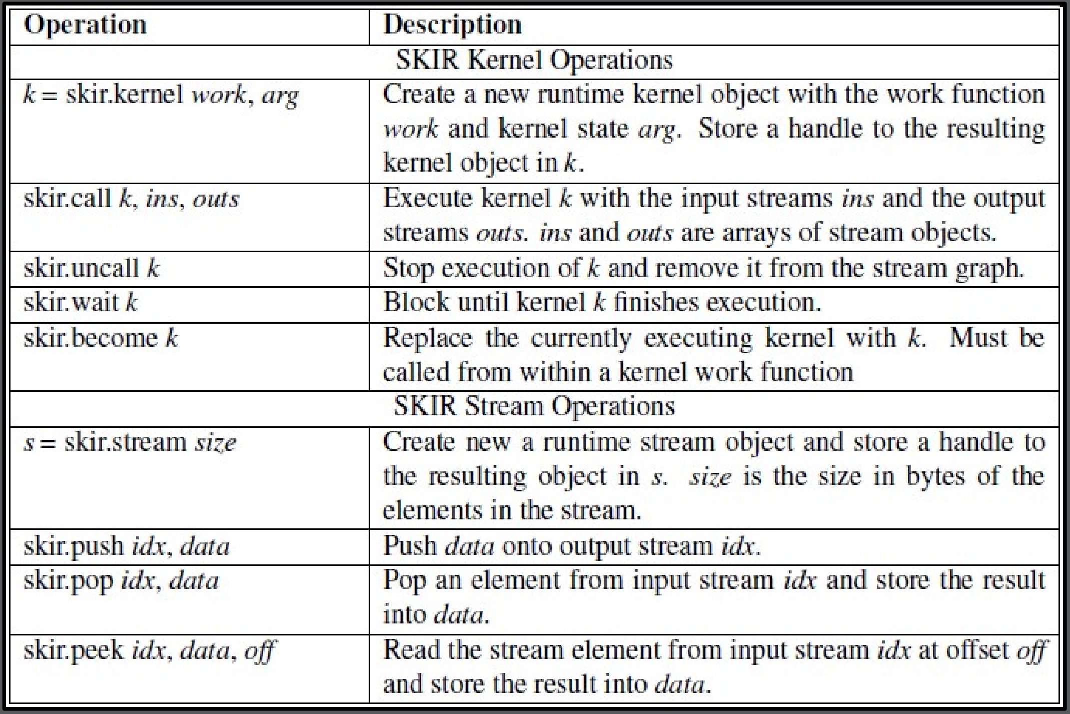
- Organized as JIT Compiler
  - for performance portability
  - for dynamic program graphs
  - for dynamic optimization
- SKIR intrinsics for LLVM
  - stream communication
  - static or dynamic stream graph manipulation

SKIR Example

- SKIR pseudo-code
- Construct and execute a 4 stage pipeline

```
bool PRODUCER (int *state, void *ins[], void *outs[])
{
    *state = *state + 1
    skir.push(0, state)
    return false
}

bool ADDER (int *state, void *ins[], void *outs[]) {
    int data
    skir.pop(0, &data)
    data += *state
    skir.push(0, &data)
    return false
}

bool CONSUMER (int *state, void *ins[], void *outs[])
{
    int data
    if (*state == 0) return true
    skir.pop(0, &data)
    print(data)
    *state = *state - 1
    return false
}
```

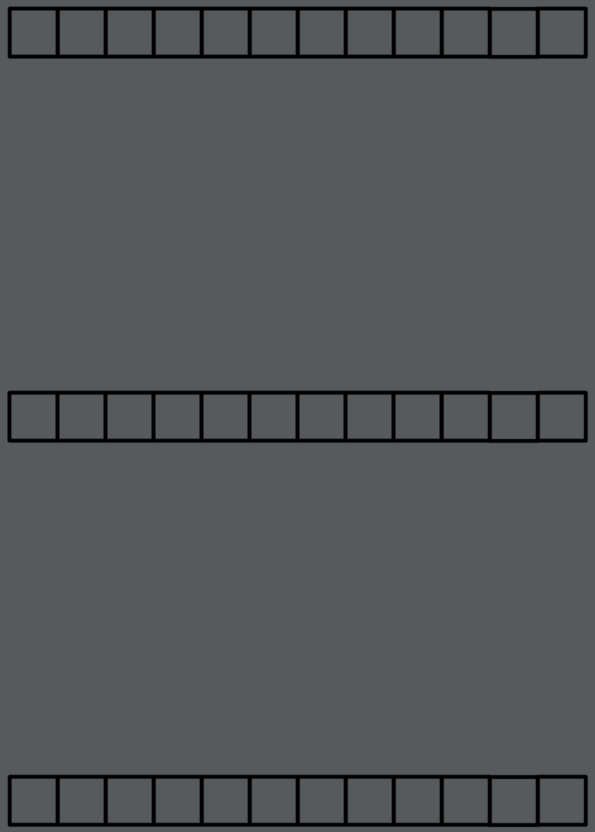
```
main()
{
    int counter = 0
    int limit = 20
    int one = 1
    int neg = -1

    stream Q1[2], Q2[2], Q3[2]
    kernel K1, K2, K3, K4

    Q1[0] = skir.stream(sizeof(int)); Q1[1] = 0
    Q2[0] = skir.stream(sizeof(int)); Q2[1] = 0
    Q3[0] = skir.stream(sizeof(int)); Q3[1] = 0

    K1 = skir.kernel(PRODUCER, &counter)
    K2 = skir.kernel(ADDER, &one)
    K3 = skir.kernel(ADDER, &neg)
    K4 = skir.kernel(CONSUMER, &limit)

    skir.call(K1, NULL, Q1)
    skir.call(K2, Q1, Q2)
    skir.call(K3, Q2, Q3)
    skir.call(K4, Q3, NULL)
    skir.wait(K4)
}
```



Operation	Description
	SKIR Kernel Operations
k = skir.kernel *work*, *arg*	Create a new runtime kernel object with the work function *work* and kernel state *arg*. Store a handle to the resulting kernel object in k.
skir.call k, *ins*, *outs*	Execute kernel k with the input streams *ins* and the output streams *outs*. *ins* and *outs* are arrays of stream objects.
skir.uncall k	Stop execution of k and remove it from the stream graph.
skir.wait k	Block until kernel k finishes execution.
skir.become k	Replace the currently executing kernel with k. Must be called from within a kernel work function
	SKIR Stream Operations
s = skir.stream *size*	Create new a runtime stream object and store a handle to the resulting object in s. *size* is the size in bytes of the elements in the stream.
skir.push *idx*, *data*	Push *data* onto output stream *idx*.
skir.pop *idx*, *data*	Pop an element from input stream *idx* and store the result into *data*.
skir.peek *idx*, *data*, *off*	Read the stream element from input stream *idx* at offset *off* and store the result into *data*.

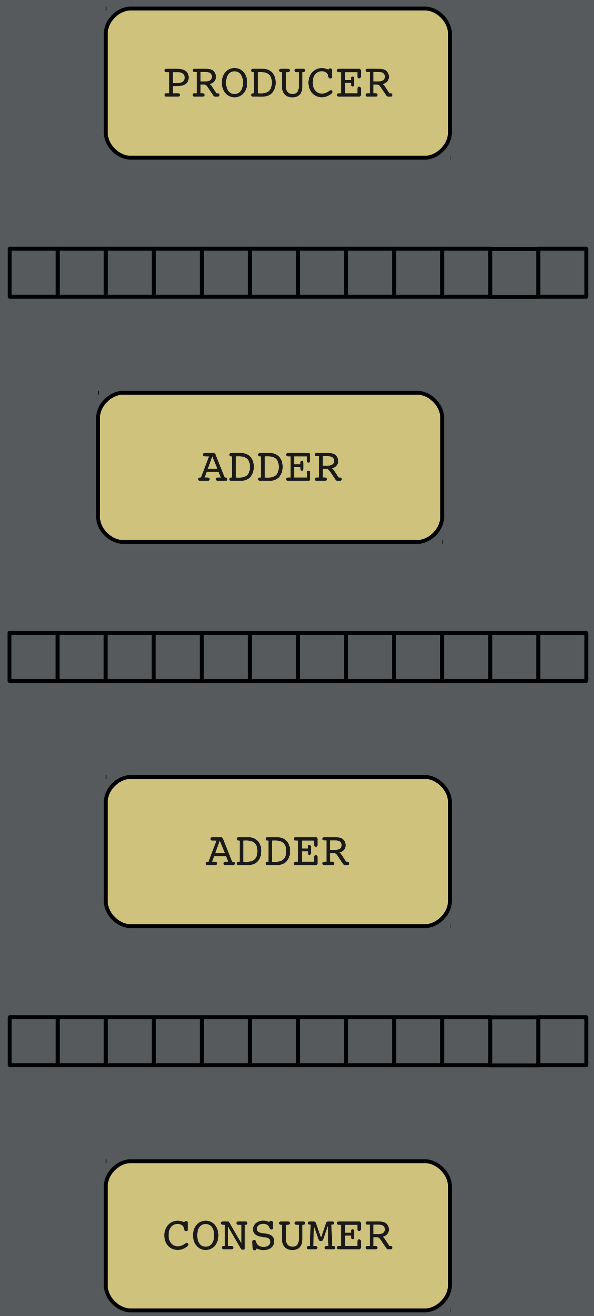
```
bool PRODUCER (int *state, void *ins[], void *outs[])
{
    *state = *state + 1
    skir.push(0, state)
    return false
}

bool ADDER (int *state, void *ins[], void *outs[]) {
    int data
    skir.pop(0, &data)
    data += *state
    skir.push(0, &data)
    return false
}

bool CONSUMER (int *state, void *ins[], void *outs[])
{
    int data
    if (*state == 0) return true
    skir.pop(0, &data)
    print(data)
    *state = *state - 1
    return false
}
```

```
main()
{
    int counter = 0
    int limit = 20
    int one = 1
    int neg = -1

    stream Q1[2], Q2[2], Q3[2]
    kernel K1, K2, K3, K4

    Q1[0] = skir.stream(sizeof(int)); Q1[1] = 0
    Q2[0] = skir.stream(sizeof(int)); Q2[1] = 0
    Q3[0] = skir.stream(sizeof(int)); Q3[1] = 0

    K1 = skir.kernel(PRODUCER, &counter)
    K2 = skir.kernel(ADDER, &one)
    K3 = skir.kernel(ADDER, &neg)
    K4 = skir.kernel(CONSUMER, &limit)

    skir.call(K1, NULL, Q1)
    skir.call(K2, Q1, Q2)
    skir.call(K3, Q2, Q3)
    skir.call(K4, Q3, NULL)
    skir.wait(K4)
}
```



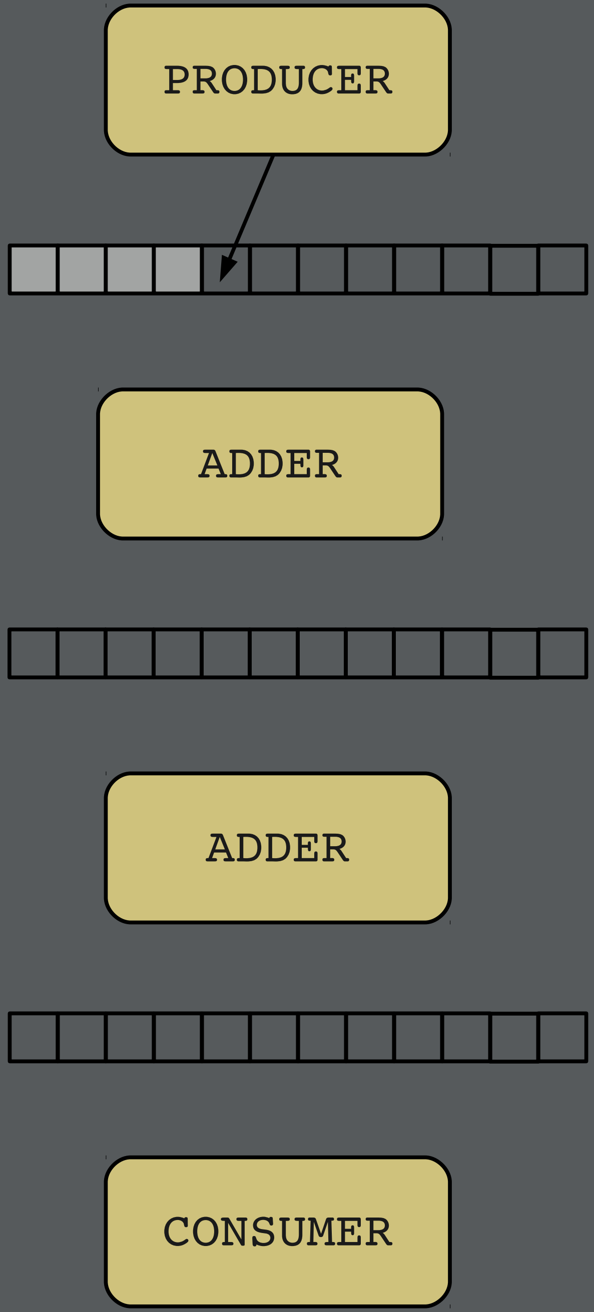
```
bool PRODUCER (int *state, void *ins[], void *outs[])
{
    *state = *state + 1
    skir.push(0, state)
    return false
}

bool ADDER (int *state, void *ins[], void *outs[]) {
    int data
    skir.pop(0, &data)
    data += *state
    skir.push(0, &data)
    return false
}

bool CONSUMER (int *state, void *ins[], void *outs[])
{
    int data
    if (*state == 0) return true
    skir.pop(0, &data)
    print(data)
    *state = *state - 1
    return false
}
```

```
main()
{
    int counter = 0
    int limit = 20
    int one = 1
    int neg = -1

    stream Q1[2], Q2[2], Q3[2]
    kernel K1, K2, K3, K4

    Q1[0] = skir.stream(sizeof(int)); Q1[1] = 0
    Q2[0] = skir.stream(sizeof(int)); Q2[1] = 0
    Q3[0] = skir.stream(sizeof(int)); Q3[1] = 0

    K1 = skir.kernel(PRODUCER, &counter)
    K2 = skir.kernel(ADDER, &one)
    K3 = skir.kernel(ADDER, &neg)
    K4 = skir.kernel(CONSUMER, &limit)

    skir.call(K1, NULL, Q1)
    skir.call(K2, Q1, Q2)
    skir.call(K3, Q2, Q3)
    skir.call(K4, Q3, NULL)
    skir.wait(K4)
}
```



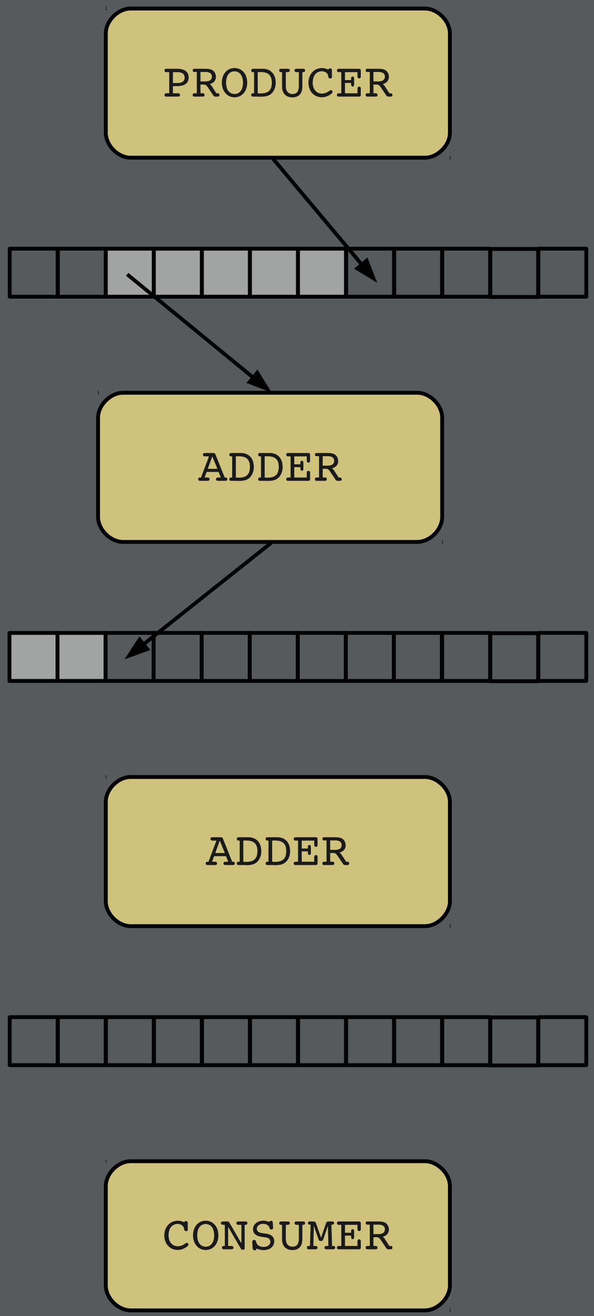
```
bool PRODUCER (int *state, void *ins[], void *outs[])
{
    *state = *state + 1
    skir.push(0, state)
    return false
}

bool ADDER (int *state, void *ins[], void *outs[]) {
    int data
    skir.pop(0, &data)
    data += *state
    skir.push(0, &data)
    return false
}

bool CONSUMER (int *state, void *ins[], void *outs[])
{
    int data
    if (*state == 0) return true
    skir.pop(0, &data)
    print(data)
    *state = *state - 1
    return false
}
```

```
main()
{
    int counter = 0
    int limit = 20
    int one = 1
    int neg = -1

    stream Q1[2], Q2[2], Q3[2]
    kernel K1, K2, K3, K4

    Q1[0] = skir.stream(sizeof(int)); Q1[1] = 0
    Q2[0] = skir.stream(sizeof(int)); Q2[1] = 0
    Q3[0] = skir.stream(sizeof(int)); Q3[1] = 0

    K1 = skir.kernel(PRODUCER, &counter)
    K2 = skir.kernel(ADDER, &one)
    K3 = skir.kernel(ADDER, &neg)
    K4 = skir.kernel(CONSUMER, &limit)

    skir.call(K1, NULL, Q1)
    skir.call(K2, Q1, Q2)
    skir.call(K3, Q2, Q3)
    skir.call(K4, Q3, NULL)
    skir.wait(K4)
}
```



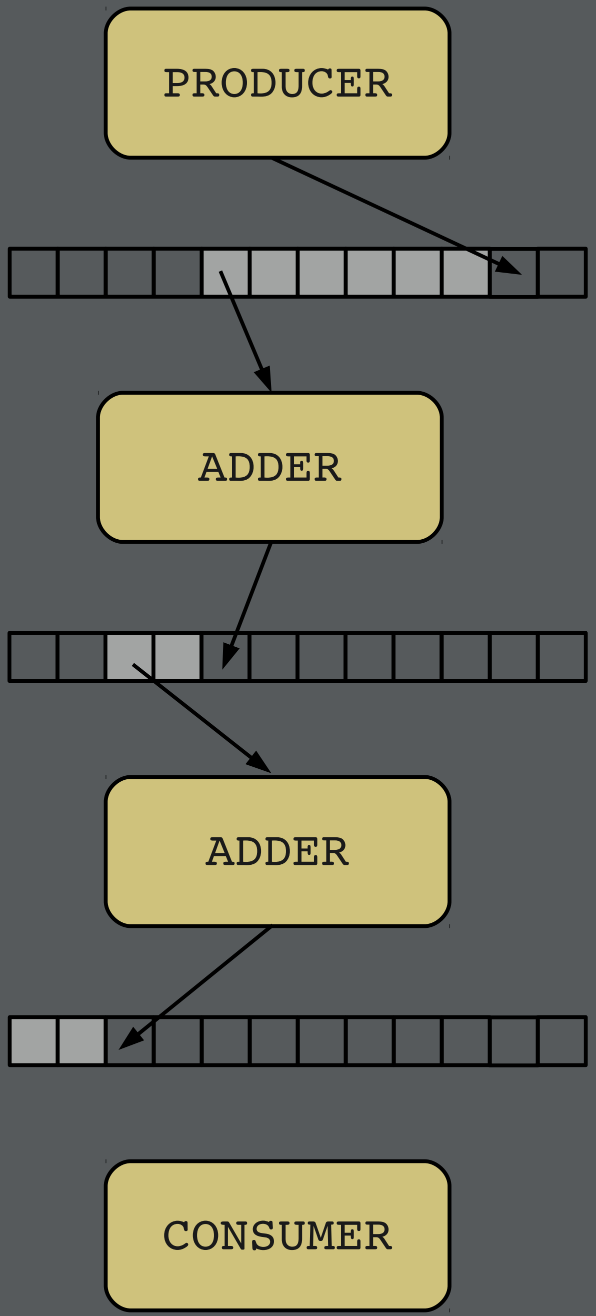
```
bool PRODUCER (int *state, void *ins[], void *outs[])
{
    *state = *state + 1
    skir.push(0, state)
    return false
}

bool ADDER (int *state, void *ins[], void *outs[]) {
    int data
    skir.pop(0, &data)
    data += *state
    skir.push(0, &data)
    return false
}

bool CONSUMER (int *state, void *ins[], void *outs[])
{
    int data
    if (*state == 0) return true
    skir.pop(0, &data)
    print(data)
    *state = *state - 1
    return false
}
```

```
main()
{
    int counter = 0
    int limit = 20
    int one = 1
    int neg = -1

    stream Q1[2], Q2[2], Q3[2]
    kernel K1, K2, K3, K4

    Q1[0] = skir.stream(sizeof(int)); Q1[1] = 0
    Q2[0] = skir.stream(sizeof(int)); Q2[1] = 0
    Q3[0] = skir.stream(sizeof(int)); Q3[1] = 0

    K1 = skir.kernel(PRODUCER, &counter)
    K2 = skir.kernel(ADDER, &one)
    K3 = skir.kernel(ADDER, &neg)
    K4 = skir.kernel(CONSUMER, &limit)

    skir.call(K1, NULL, Q1)
    skir.call(K2, Q1, Q2)
    skir.call(K3, Q2, Q3)
    skir.call(K4, Q3, NULL)
    skir.wait(K4)
}
```



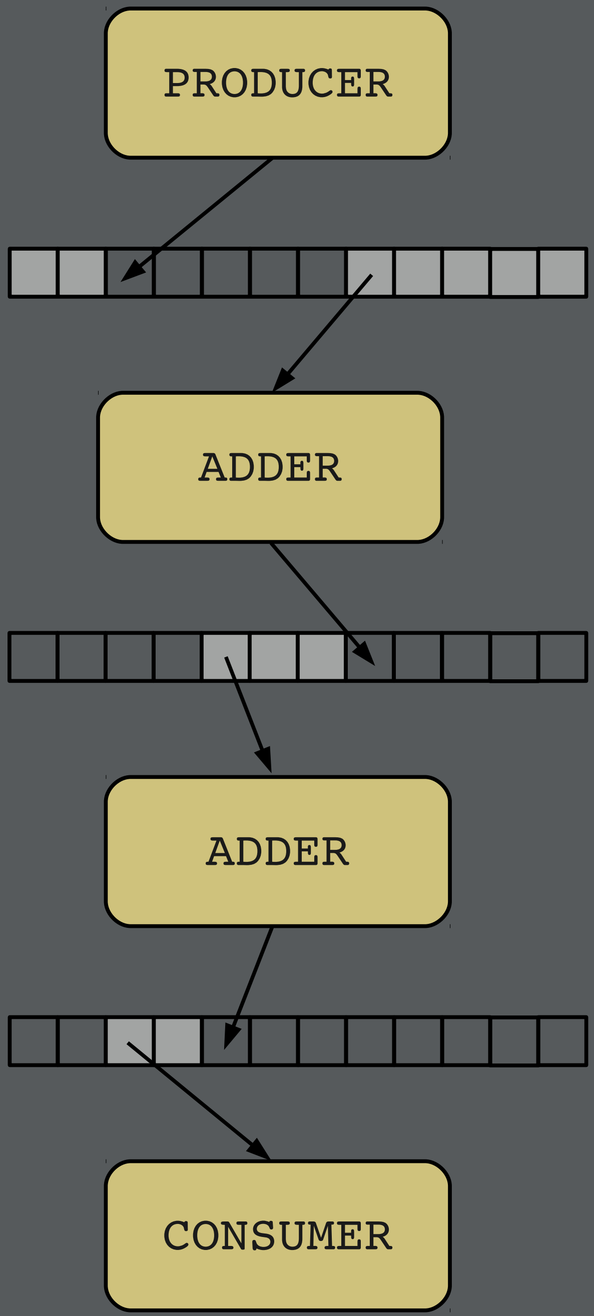
```
bool PRODUCER (int *state, void *ins[], void *outs[])
{
    *state = *state + 1
    skir.push(0, state)
    return false
}

bool ADDER (int *state, void *ins[], void *outs[]) {
    int data
    skir.pop(0, &data)
    data += *state
    skir.push(0, &data)
    return false
}

bool CONSUMER (int *state, void *ins[], void *outs[])
{
    int data
    if (*state == 0) return true
    skir.pop(0, &data)
    print(data)
    *state = *state - 1
    return false
}
```

```
main()
{
    int counter = 0
    int limit = 20
    int one = 1
    int neg = -1

    stream Q1[2], Q2[2], Q3[2]
    kernel K1, K2, K3, K4

    Q1[0] = skir.stream(sizeof(int)); Q1[1] = 0
    Q2[0] = skir.stream(sizeof(int)); Q2[1] = 0
    Q3[0] = skir.stream(sizeof(int)); Q3[1] = 0

    K1 = skir.kernel(PRODUCER, &counter)
    K2 = skir.kernel(ADDER, &one)
    K3 = skir.kernel(ADDER, &neg)
    K4 = skir.kernel(CONSUMER, &limit)

    skir.call(K1, NULL, Q1)
    skir.call(K2, Q1, Q2)
    skir.call(K3, Q2, Q3)
    skir.call(K4, Q3, NULL)
    skir.wait(K4)
}
```



```
bool PRODUCER (int *state, void *ins[], void *outs[])
{
    *state = *state + 1
    skir.push(0, state)
    return false
}

bool ADDER (int *state, void *ins[], void *outs[]) {
    int data
    skir.pop(0, &data)
    data += *state
    skir.push(0, &data)
    return false
}

bool CONSUMER (int *state, void *ins[], void *outs[])
{
    int data
    if (*state == 0) return true
    skir.pop(0, &data)
    print(data)
    *state = *state - 1
    return false
}
```

```
main()
{
    int counter = 0
    int limit = 20
    int one = 1
    int neg = -1

    stream Q1[2], Q2[2], Q3[2]
    kernel K1, K2, K3, K4

    Q1[0] = skir.stream(sizeof(int)); Q1[1] = 0
    Q2[0] = skir.stream(sizeof(int)); Q2[1] = 0
    Q3[0] = skir.stream(sizeof(int)); Q3[1] = 0

    K1 = skir.kernel(PRODUCER, &counter)
    K2 = skir.kernel(ADDER, &one)
    K3 = skir.kernel(ADDER, &neg)
    K4 = skir.kernel(CONSUMER, &limit)

    skir.call(K1, NULL, Q1)
    skir.call(K2, Q1, Q2)
    skir.call(K3, Q2, Q3)
    skir.call(K4, Q3, NULL)
    skir.wait(K4)
}
```



SKIR as a compiler target: C

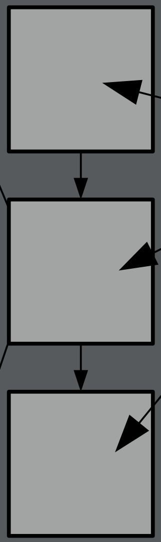
```
int
subtracter_work(void *state, skir_stream_ptr_t *ins, skir_stream_ptr_t *outs)
{
    float f0;
    float f1;
    __SKIR_pop(0, &f0);
    __SKIR_pop(0, &f1);
    f0 = f1 - f0;
    __SKIR_push(0, &f0);
    return 0;
}

skir_stream_ptr_t
build_band_pass_filter(skir_stream_ptr_t src,
                       float rate, float low, float high, int taps)
{
    skir_stream_ptr_t ins[2] = {0};
    skir_stream_ptr_t outs[2] = {0};

    src = build_bpf_core(src, rate, low, high, taps);
    skir_kernel_ptr_t sub = __SKIR_kernel((void*)subtracter_work, 0);
    ins[0] = src;
    outs[0] = __SKIR_stream(sizeof(float));
    __SKIR_call(sub, ins, outs);

    return outs[0];
}
```

One-to-one mapping of SKIR
operations to C intrinsics

SKIR as a compiler target: C++

```cpp
#include <SKIR.hpp>

class CalculateForces : public Kernel<CalculateForces>
{
public:
    float m_pos_rd[4*NBODIES];
    float m_softeningSquared;

    CalculateForces(float &softeningSquared)
            : m_softeningSquared(softeningSquared)
    {   }

    void interaction(float *accel, int pos0, int pos1)  {
            // compute acceleration
            ...
    }

    static int work(CalculateForces *me, StreamPtr ins[],
                                          StreamPtr outs[])
    {
            Stream<int> in(ins,0);
            Stream<float> out(outs,0);

            float force[3] = {0.0f,0.0f,0.0f};

            int i = in.pop()*4;
            int N = in.pop()*4;

            for (int j=0; j<N; j+=16) {
                me->interaction(force, j, i);
                me->interaction(force, j+4, i);
                me->interaction(force, j+8, i);
                me->interaction(force, j+12, i);
            }

            float f = i/4;
            out.push(f);
            out.push(force[0]);
            out.push(force[1]);
            out.push(force[2]);

            return 0;
    }
};
```

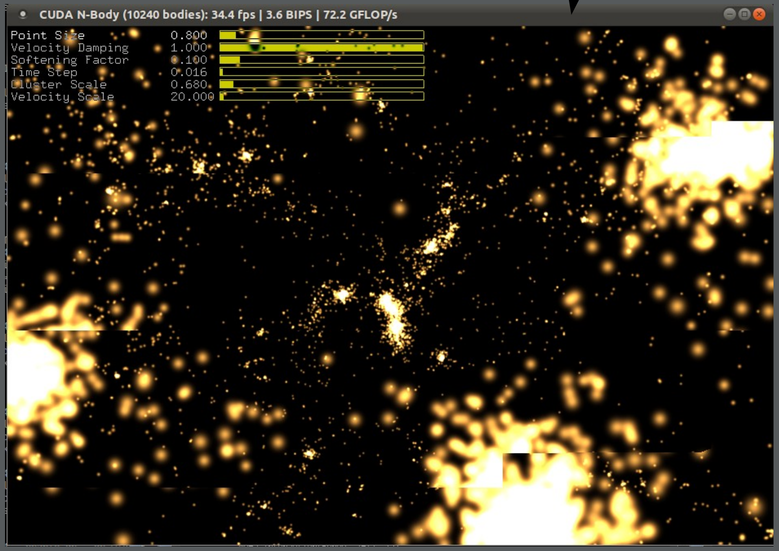
A high level C++ library maps object oriented stream parallelism onto SKIR intrinsics



stream parallelism is embedded deep inside the application

```cpp
update(..) {
                ...
    (*m_pairs)(nul0, s0);
    (*m_calc)(s0, s1);
    (*m_integ)(s1, nul1);

    m_integ->wait();
    m_calc->wait();
    m_pairs->wait();
                ...
}
```

```cpp
display()
{
    ...
    nbody->update(Δt);
    renderer >setPositions(...);
    ...
}
```



CUDA N-Body (10240 bodies): 34.4 fps | 3.6 BIPS | 72.2 GFLOP/s

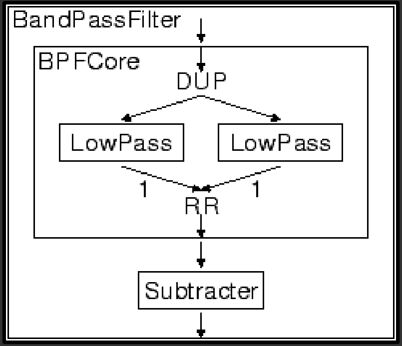
```
Point Size           0.800
Velocity Damping     1.000
Softening Factor     0.100
Time Step            0.016
Cluster Scale        0.680
Velocity Scale       20.000
```

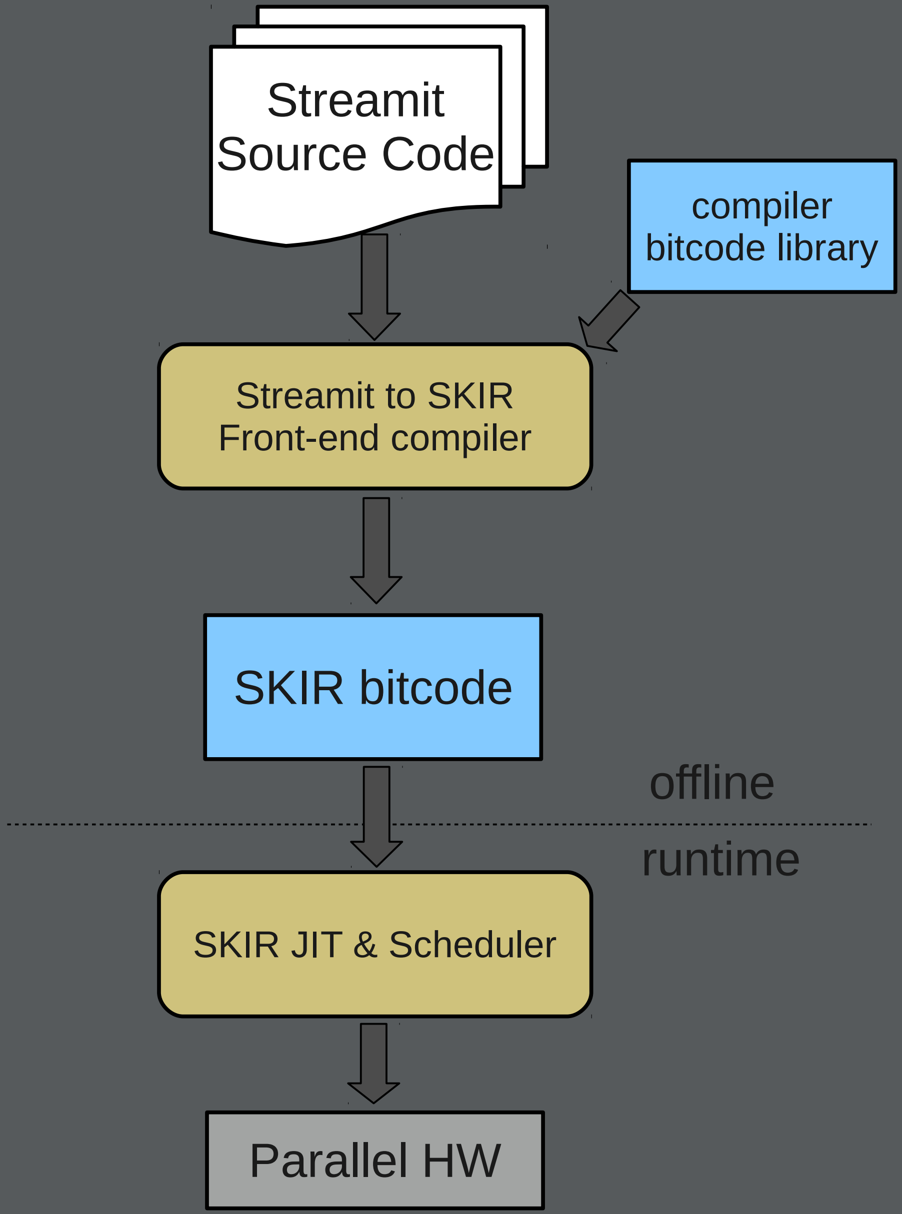
Example:
N-Body Simulation
from CUDA SDK

SKIR as a compiler target: StreamIt

- Stream Language from MIT
  - Independent Filters
  - FIFO Streams
- Synchronous Data Flow
  - Fixed I/O Rates
  - Fixed stream graph structure

```
float->float pipeline BandPassFilter(float rate, float low, float high, int taps)
{
  add BPFCore(rate, low, high, taps);
  add Subtracter();
}

float->float splitjoin BPFCore (float rate, float low, float high, int taps)
{
  split duplicate;
  add LowPassFilter(rate, low, taps, 0);
  add LowPassFilter(rate, high, taps, 0);
  join roundrobin;
}

float->float filter Subtracter
{
  work pop 2 push 1 {
    push(peek(1) - peek(0));
    pop(); pop();
  }
}
```





Streamit Source Code

compiler bitcode library

Streamit to SKIR Front-end compiler

SKIR bitcode

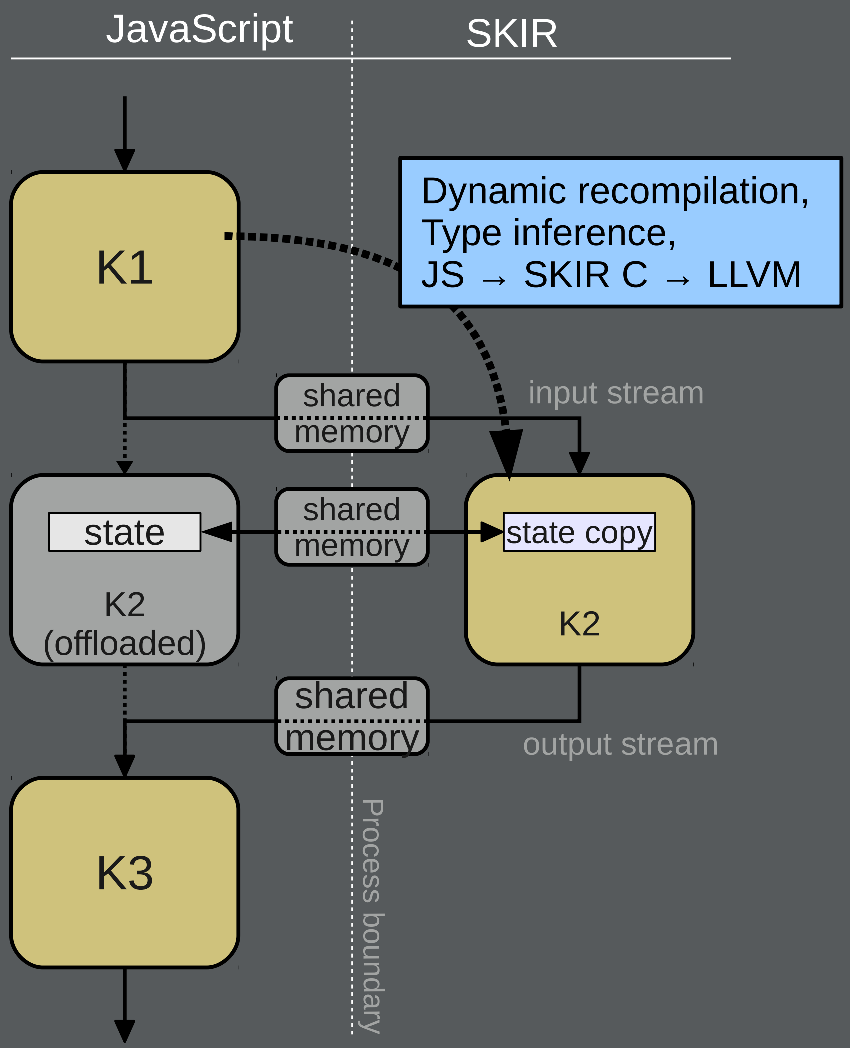
offline

runtime

SKIR JIT & Scheduler

Parallel HW

SKIR as a compiler target: JavaScript



JavaScript

SKIR

K1

Dynamic recompilation,
Type inference,
JS → SKIR C → LLVM

shared memory

input stream

state

K2 (offloaded)

shared memory

state copy

K2

shared memory

output stream

K3

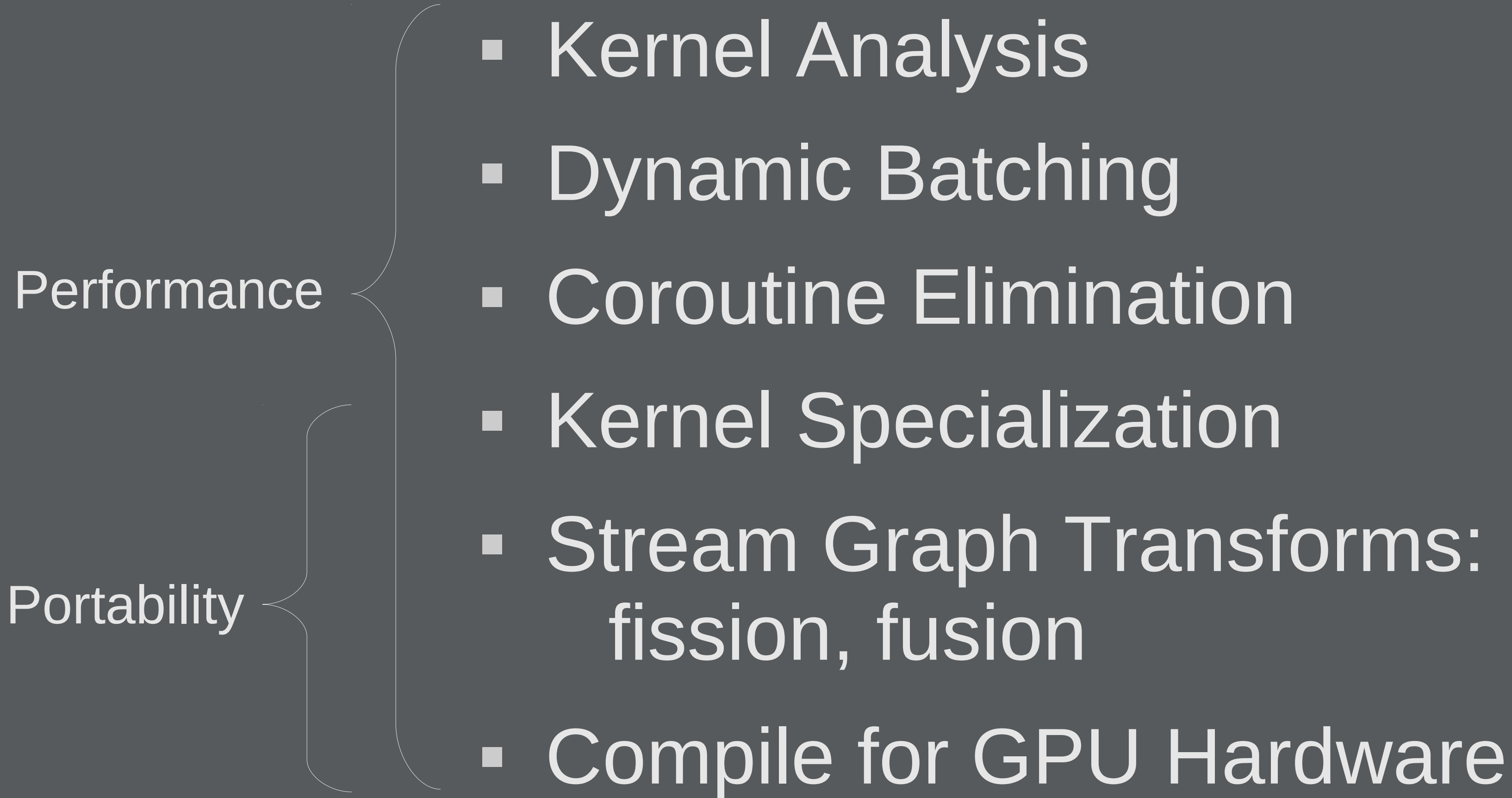
Process boundary

Sluice: SKIR based acceleration of StreamIt style stream parallelism for the node.js/V8 JavaScript environment

```
function Adder(arg) {
    this.a = arg;
    this.work = function() {
        var e = this.pop();
        e = e + this.a;
        this.push(e);
        return false;
    }
}
var a0 = new Adder(1);
var a1 = new Adder(1);
var a2 = new Adder(1);
var sj = Sluice.SplitRR(1,a0,a1,a2).JoinRR(1);
var p = Sluice.Pipeline(new Count(10),
                        sj,
                        new Printer());


> p.run()
1 2 3 4 5 6 7 8 9 10
```

J. Fifield and D. Grunwald. *A Methodology for Fine-Grained Parallelization of JavaScript Applications.* LCPC 2011

Compiling SKIR: Overview



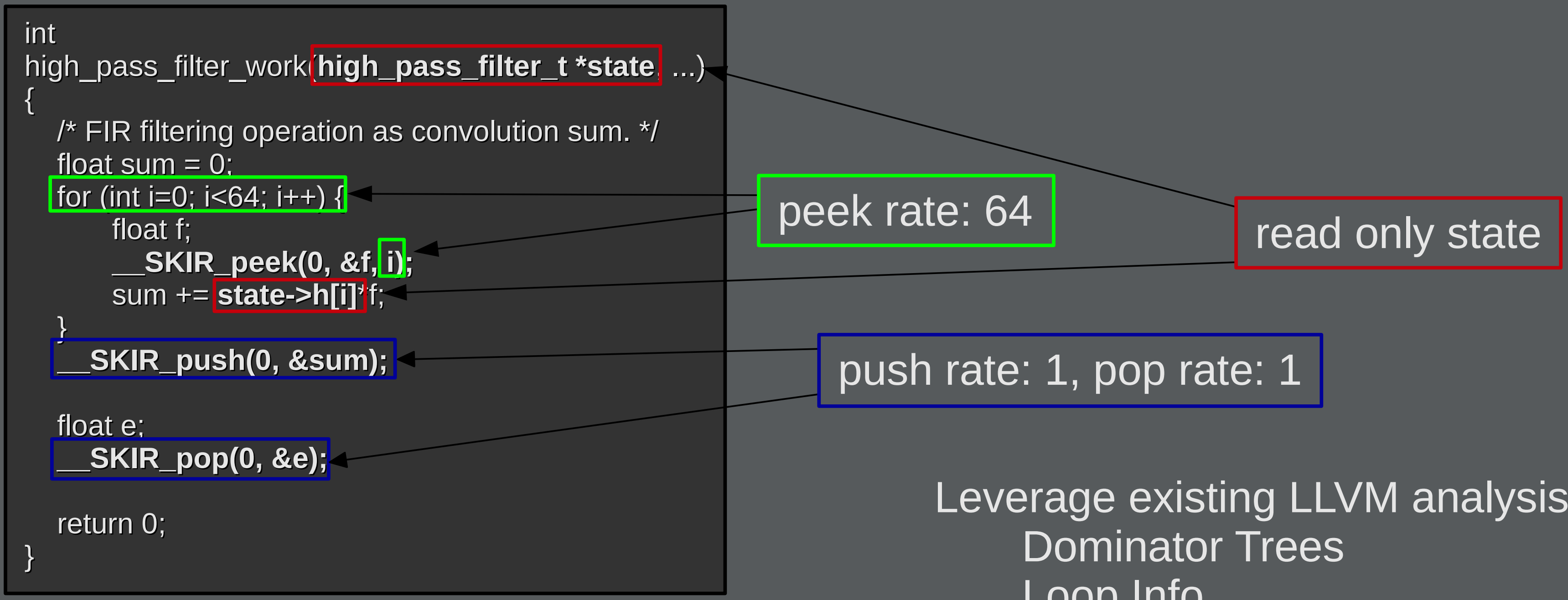
Performance

Portability

- Kernel Analysis
- Dynamic Batching
- Coroutine Elimination
- Kernel Specialization
- Stream Graph Transforms: fission, fusion
- Compile for GPU Hardware

Compiling SKIR: Kernel Analysis

Attempt to extract SDF semantics from arbitrary kernels
- push, pop, peek rates
- data parallel vs. stateful



```
int
high_pass_filter_work(high_pass_filter_t *state, ...)
{
   /* FIR filtering operation as convolution sum. */
   float sum = 0;
   for (int i=0; i<64; i++) {
         float f;
         __SKIR_peek(0, &f, i);
         sum += state->h[i]*f;
   }
   __SKIR_push(0, &sum);

   float e;
   __SKIR_pop(0, &e);

   return 0;
}
```

peek rate: 64

read only state

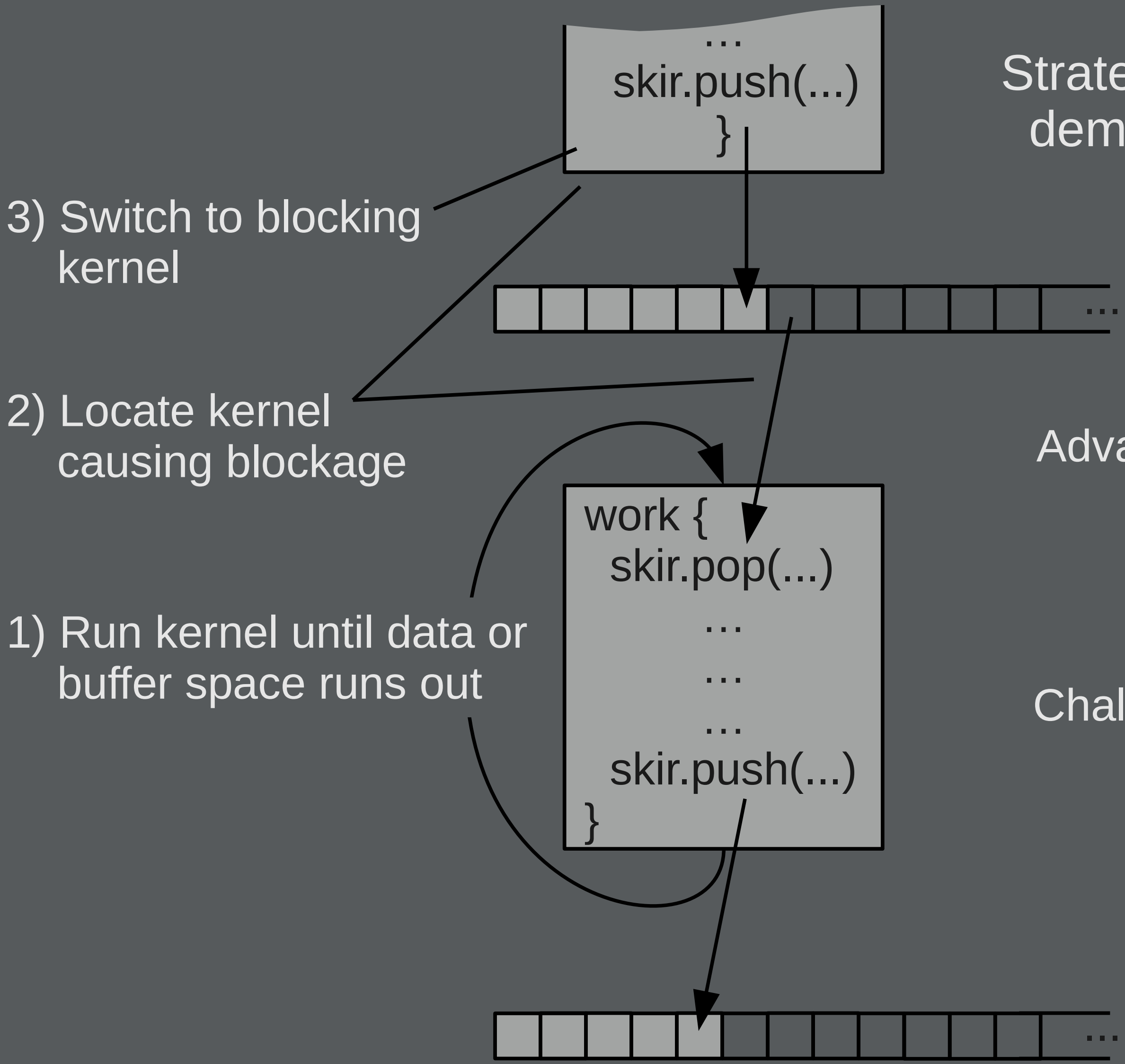
push rate: 1, pop rate: 1

C version of a kernel from StreamIt
channel vocoder benchmark

Leverage existing LLVM analysis:
- Dominator Trees
- Loop Info
- Scalar Evolution
- Def/Use Information
- Stack/alloca Information

Scheduling SKIR:

Demand and data driven execution



```
…
skir.push(...)
}
```

3) Switch to blocking kernel

2) Locate kernel causing blockage

1) Run kernel until data or buffer space runs out

```
work {
  skir.pop(...)
  …
  …
  …
  skir.push(...)
}
```

Strategy: Schedule kernels by following demand for data and buffer space

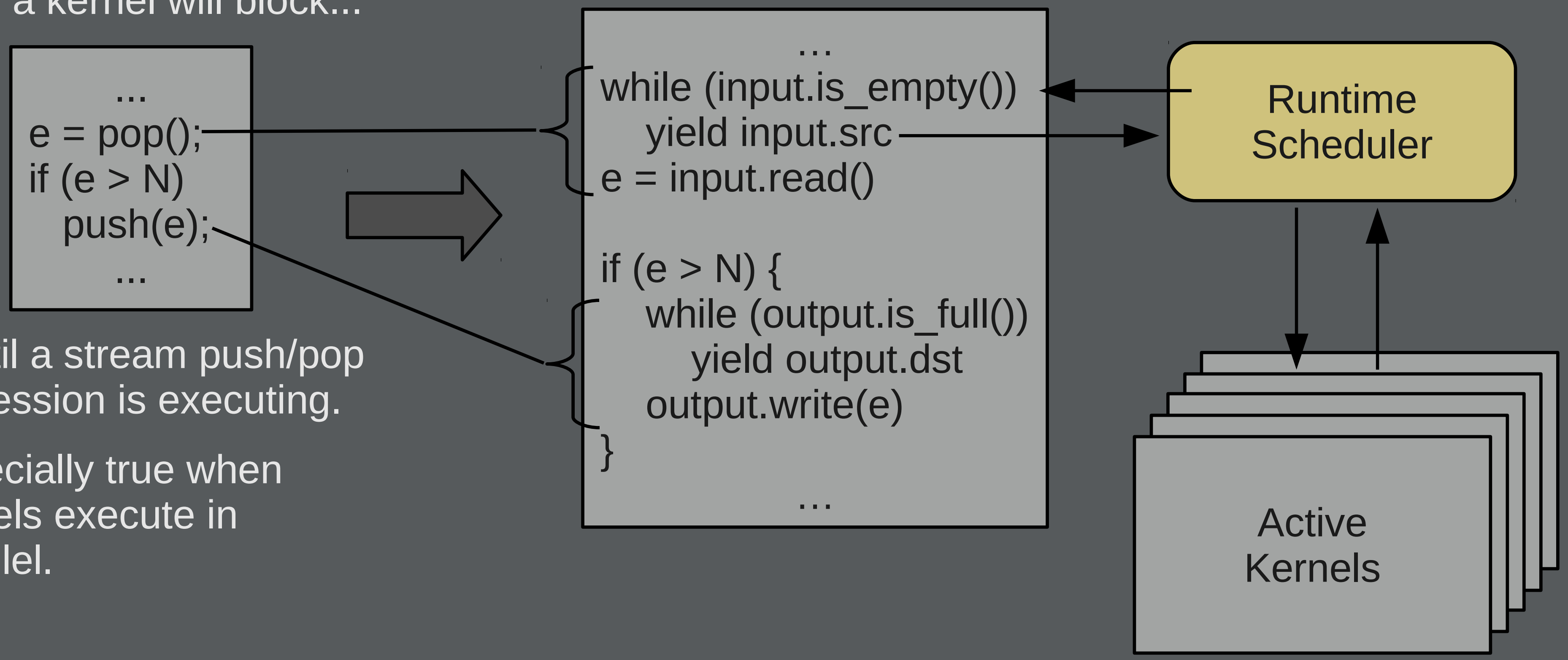
Advantages:
  - Doesn't require global task queues
  - Doesn't access global program structure
  - Attempts to preserve locality

Challenges:
  - Avoiding unnecessary execution
  - Making it fast

Coroutine Scheduling:

How SKIR creates demand driven execution

In general, we cannot know when a kernel will block...



```
...
e = pop();
if (e > N)
   push(e);
...
```

...until a stream push/pop expression is executing.

Especially true when kernels execute in parallel.

During code generation, we transform kernels to coroutines by specializing stream communication.

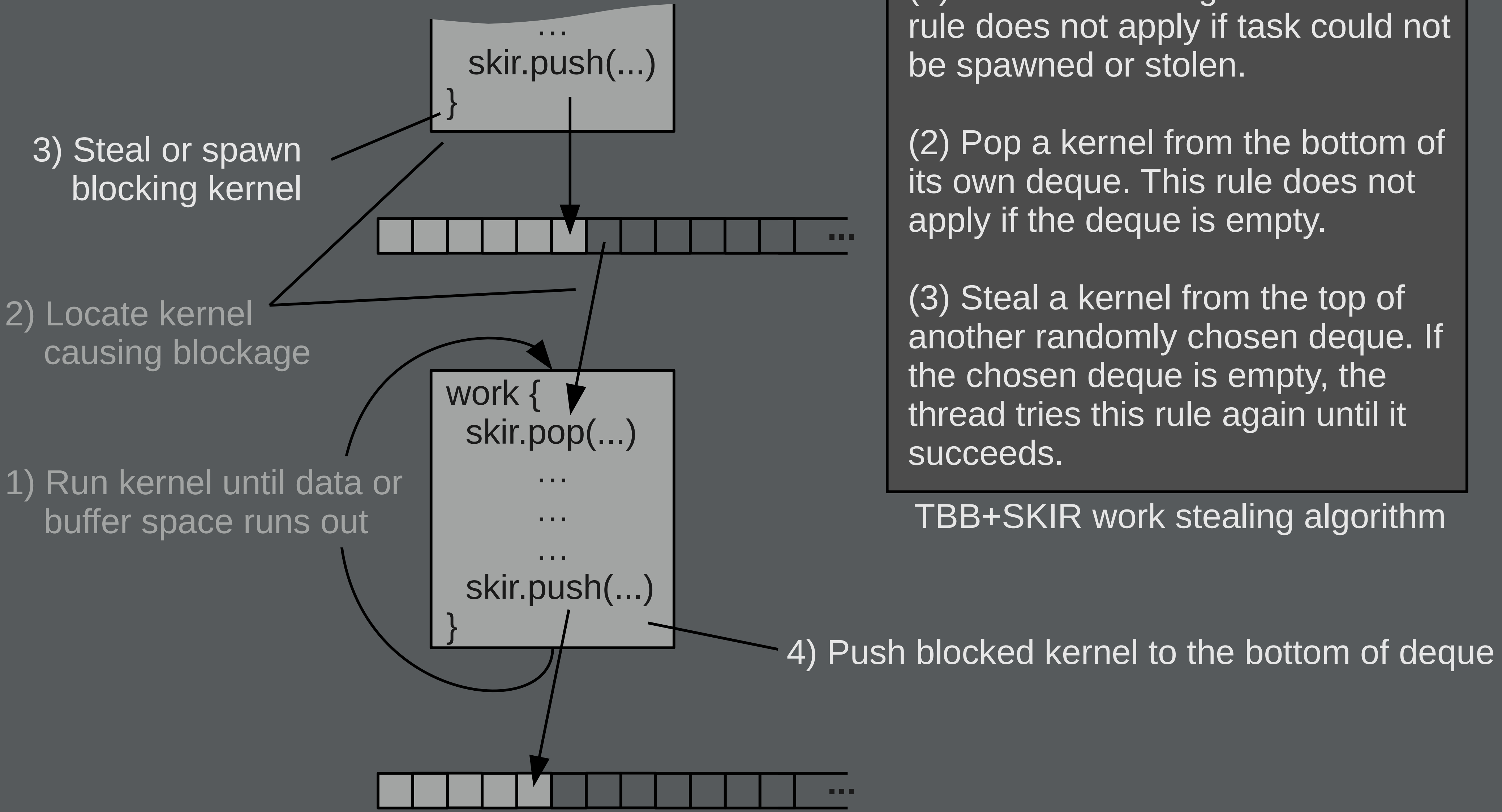
```
...
while (input.is_empty())
    yield input.src
e = input.read()

if (e > N) {
    while (output.is_full())
        yield output.dst
    output.write(e)
}
...
```

Runtime Scheduler

Active Kernels

Scheduling SKIR:

Obtaining parallel execution with task stealing



```
…
  skir.push(...)
}
```

3) Steal or spawn blocking kernel

2) Locate kernel causing blockage

1) Run kernel until data or buffer space runs out

```
work {
  skir.pop(...)
  …
  …
  …
  skir.push(...)
}
```

4) Push blocked kernel to the bottom of deque

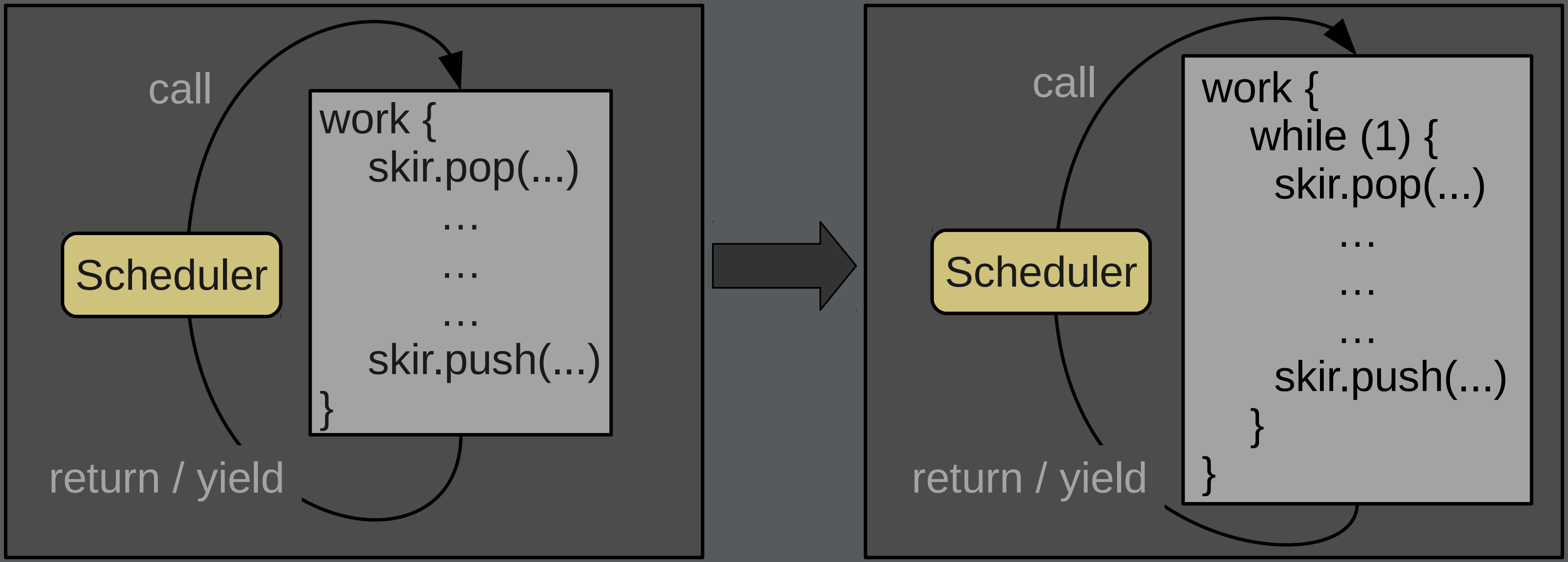
(1) Run the blocking kernel. This rule does not apply if task could not be spawned or stolen.

(2) Pop a kernel from the bottom of its own deque. This rule does not apply if the deque is empty.

(3) Steal a kernel from the top of another randomly chosen deque. If the chosen deque is empty, the thread tries this rule again until it succeeds.

TBB+SKIR work stealing algorithm

Compiling SKIR: Dynamic Batching



```
work {
    skir.pop(...)
        ...
        ...
        ...
    skir.push(...)
}
```

High overhead for small kernels

```
work {
    while (1) {
      skir.pop(...)
          ...
          ...
          ...
      skir.push(...)
    }
}
```

Run as long as data/buffer available

Compiling SKIR: Coroutine Elimination

The default coroutine code transformation is fine for coarse-grained kernels, but it has high overhead for fine-grained kernels.

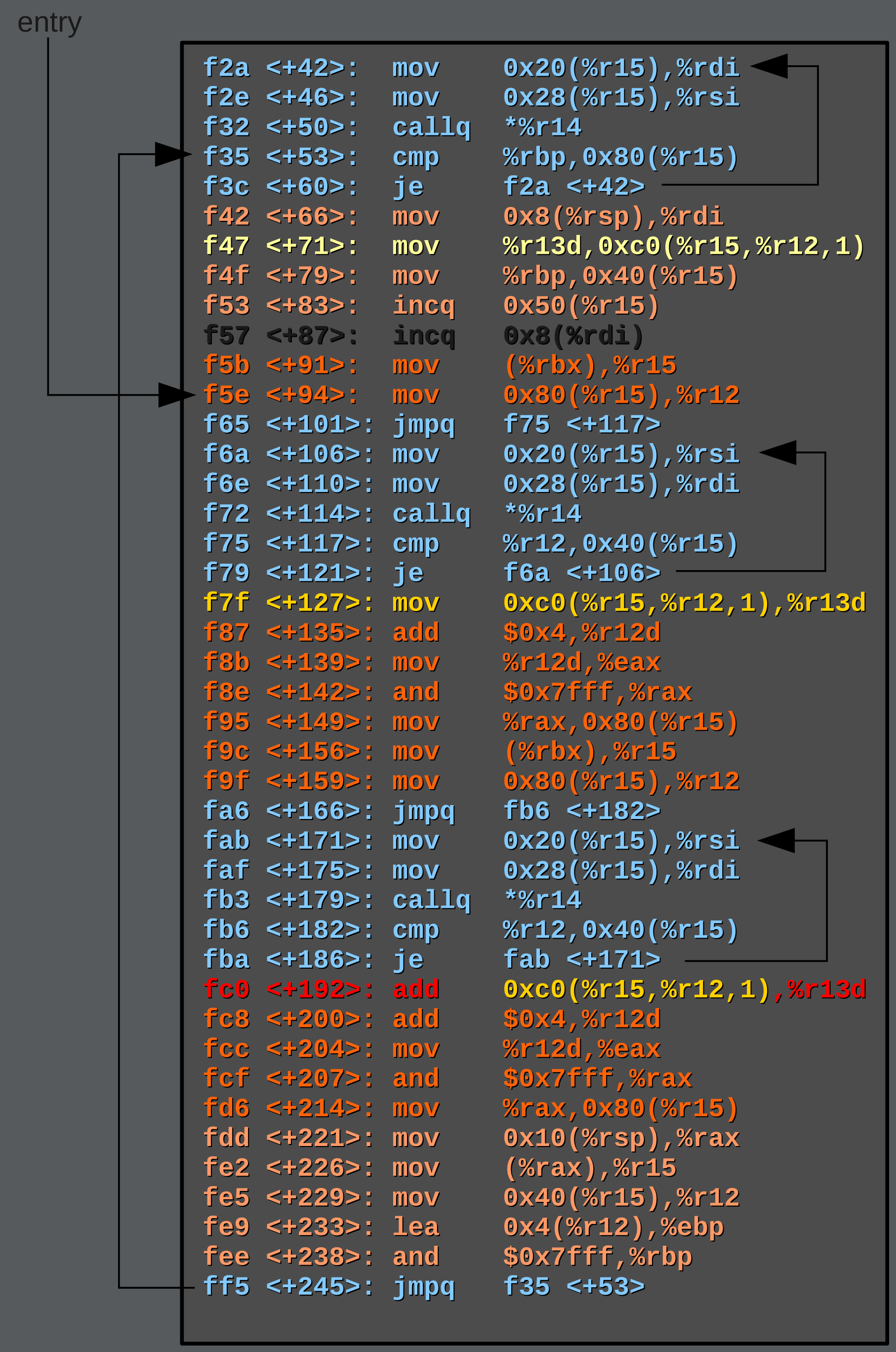
```
work(...)
{
  while(1) {
    while (input.is_empty())
      yield input.src
    e = input.read()

    do_actual_work

    while (output.is_full())
      yield output.dst
    output.write(e)
  }
}
```

Not good if “`do_actual_work`” is small

```
work(...)
{
  while (1) {
    n = niters(input, output)
    while(n--) {
      e = input.read()

      do_actual_work

      output.write(e)
    }
  }
}
```

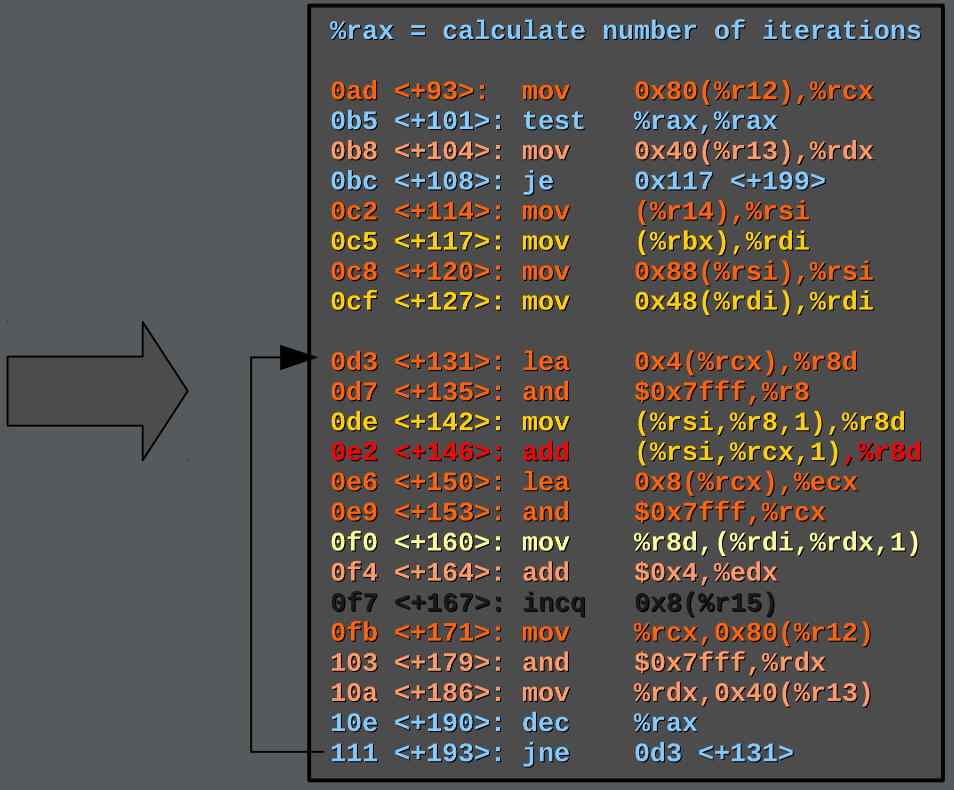
We can be smarter for kernels with fixed I/O rates

Impact of Coroutine Elimination



```
f2a <+42>:  mov    0x20(%r15),%rdi
f2e <+46>:  mov    0x28(%r15),%rsi
f32 <+50>:  callq  *%r14
f35 <+53>:  cmp    %rbp,0x80(%r15)
f3c <+60>:  je     f2a <+42>
f42 <+66>:  mov    0x8(%rsp),%rdi
f47 <+71>:  mov    %r13d,0xc0(%r15,%r12,1)
f4f <+79>:  mov    %rbp,0x40(%r15)
f53 <+83>:  incq   0x50(%r15)
f57 <+87>:  incq   0x8(%rdi)
f5b <+91>:  mov    (%rbx),%r15
f5e <+94>:  mov    0x80(%r15),%r12
f65 <+101>: jmpq   f75 <+117>
f6a <+106>: mov    0x20(%r15),%rsi
f6e <+110>: mov    0x28(%r15),%rdi
f72 <+114>: callq  *%r14
f75 <+117>: cmp    %r12,0x40(%r15)
f79 <+121>: je     f6a <+106>
f7f <+127>: mov    0xc0(%r15,%r12,1),%r13d
f87 <+135>: add    $0x4,%r12d
f8b <+139>: mov    %r12d,%eax
f8e <+142>: and    $0x7fff,%rax
f95 <+149>: mov    %rax,0x80(%r15)
f9c <+156>: mov    (%rbx),%r15
f9f <+159>: mov    0x80(%r15),%r12
fa6 <+166>: jmpq   fb6 <+182>
fab <+171>: mov    0x20(%r15),%rsi
faf <+175>: mov    0x28(%r15),%rdi
fb3 <+179>: callq  *%r14
fb6 <+182>: cmp    %r12,0x40(%r15)
fba <+186>: je     fab <+171>
fc0 <+192>: add    0xc0(%r15,%r12,1),%r13d
fc8 <+200>: add    $0x4,%r12d
fcc <+204>: mov    %r12d,%eax
fcf <+207>: and    $0x7fff,%rax
fd6 <+214>: mov    %rax,0x80(%r15)
fdd <+221>: mov    0x10(%rsp),%rax
fe2 <+226>: mov    (%rax),%r15
fe5 <+229>: mov    0x40(%r15),%r12
fe9 <+233>: lea    0x4(%r12),%ebp
fee <+238>: and    $0x7fff,%rbp
ff5 <+245>: jmpq   f35 <+53>
```

without



```
%rax = calculate number of iterations

0ad <+93>:  mov    0x80(%r12),%rcx
0b5 <+101>: test   %rax,%rax
0b8 <+104>: mov    0x40(%r13),%rdx
0bc <+108>: je     0x117 <+199>
0c2 <+114>: mov    (%r14),%rsi
0c5 <+117>: mov    (%rbx),%rdi
0c8 <+120>: mov    0x88(%rsi),%rsi
0cf <+127>: mov    0x48(%rdi),%rdi

0d3 <+131>: lea    0x4(%rcx),%r8d
0d7 <+135>: and    $0x7fff,%r8
0de <+142>: mov    (%rsi,%r8,1),%r8d
0e2 <+146>: add    (%rsi,%rcx,1),%r8d
0e6 <+150>: lea    0x8(%rcx),%ecx
0e9 <+153>: and    $0x7fff,%rcx
0f0 <+160>: mov    %r8d,(%rdi,%rdx,1)
0f4 <+164>: add    $0x4,%edx
0f7 <+167>: incq   0x8(%r15)
0fb <+171>: mov    %rcx,0x80(%r12)
103 <+179>: and    $0x7fff,%rdx
10a <+186>: mov    %rdx,0x40(%r13)
10e <+190>: dec    %rax
111 <+193>: jne    0d3 <+131>
```

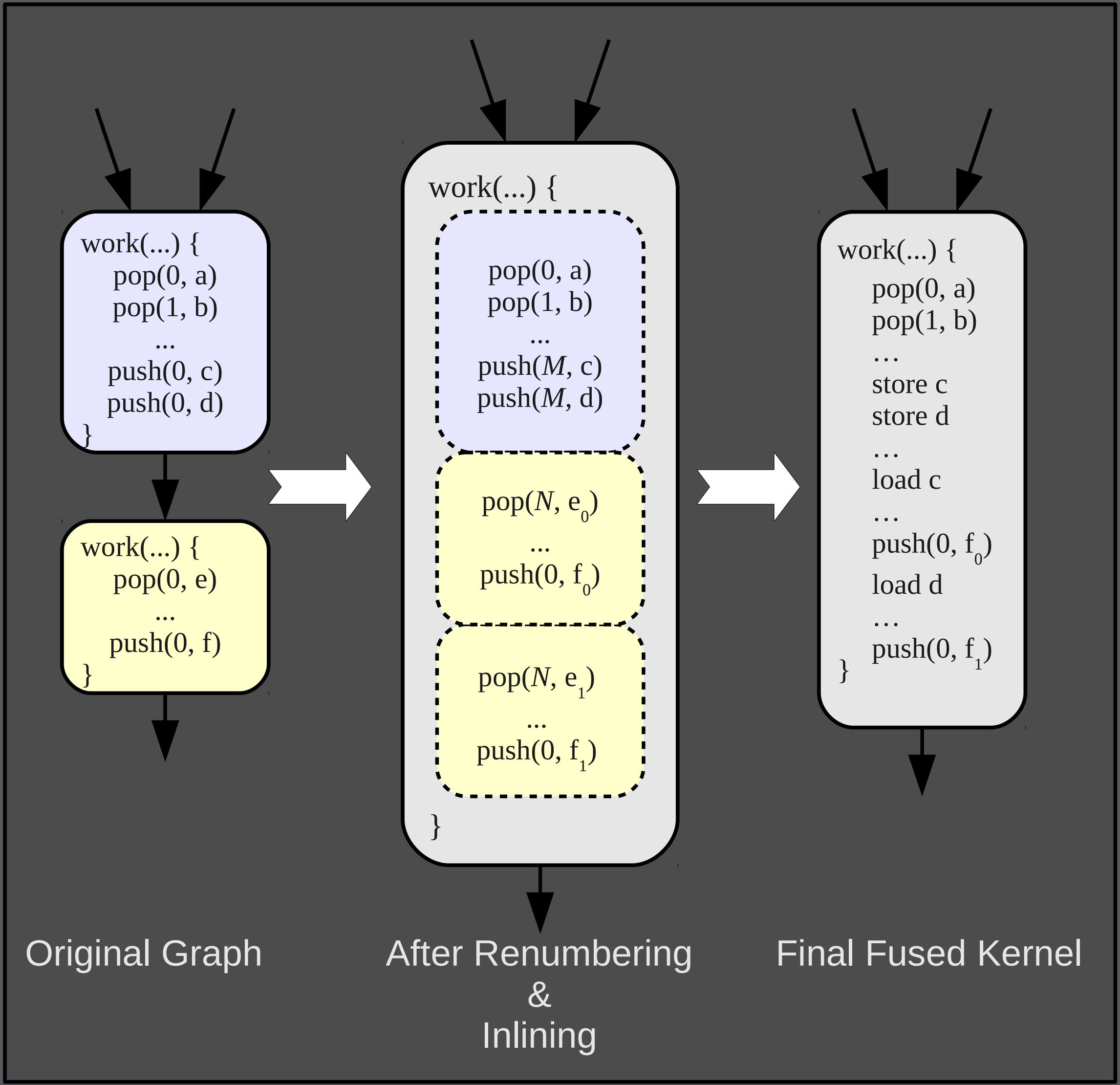
with

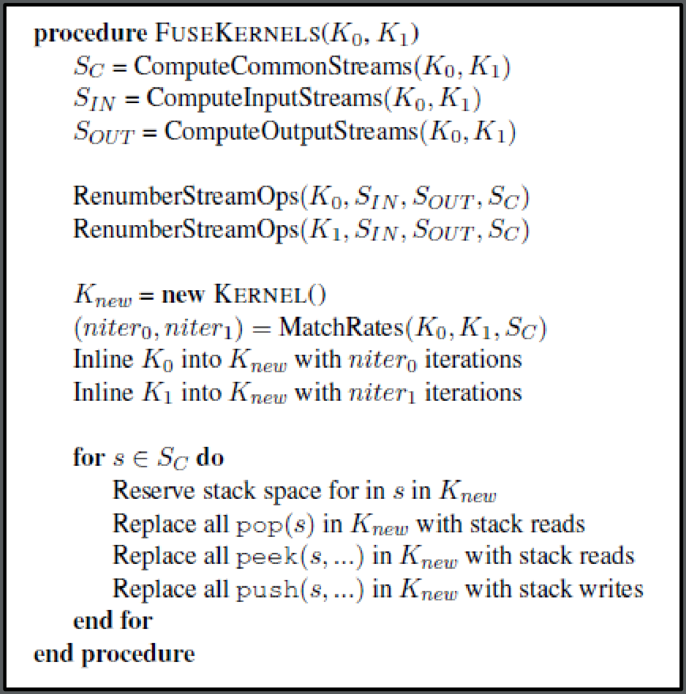
```
int -> int filter adder
{
  work {
    push(pop()+pop());
  }
}
```

input

stream read
read address calc
stream write
write address calc
kernel work (the add)
profiling
scheduler

Compiling SKIR: Kernel Fusion

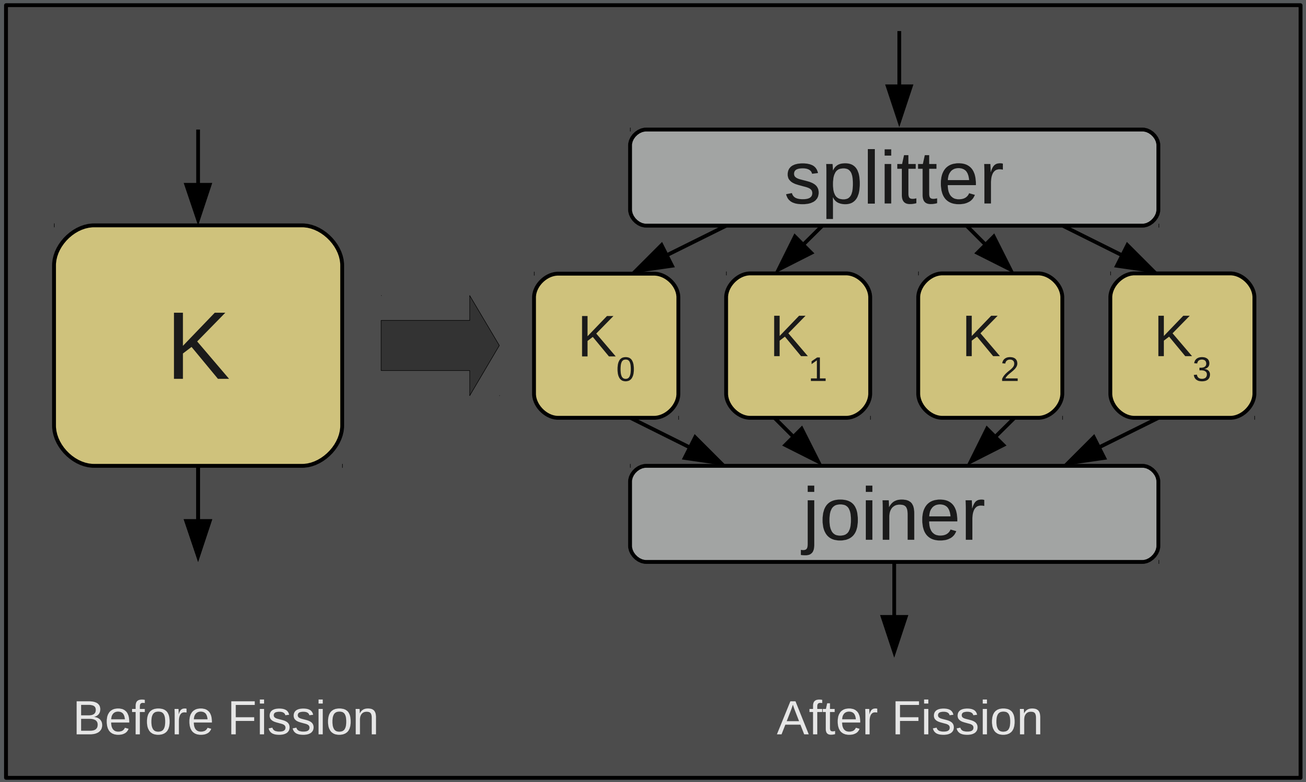




```
procedure FuseKernels($K_0$, $K_1$)
    $S_C$ = ComputeCommonStreams($K_0$, $K_1$)
    $S_{IN}$ = ComputeInputStreams($K_0$, $K_1$)
    $S_{OUT}$ = ComputeOutputStreams($K_0$, $K_1$)

    RenumberStreamOps($K_0$, $S_{IN}$, $S_{OUT}$, $S_C$)
    RenumberStreamOps($K_1$, $S_{IN}$, $S_{OUT}$, $S_C$)

    $K_{new}$ = new Kernel()
    $(niter_0, niter_1)$ = MatchRates($K_0$, $K_1$, $S_C$)
    Inline $K_0$ into $K_{new}$ with $niter_0$ iterations
    Inline $K_1$ into $K_{new}$ with $niter_1$ iterations

    for $s \in S_C$ do
        Reserve stack space for in $s$ in $K_{new}$
        Replace all pop($s$) in $K_{new}$ with stack reads
        Replace all peek($s$, ...) in $K_{new}$ with stack reads
        Replace all push($s$, ...) in $K_{new}$ with stack writes
    end for
end procedure
```

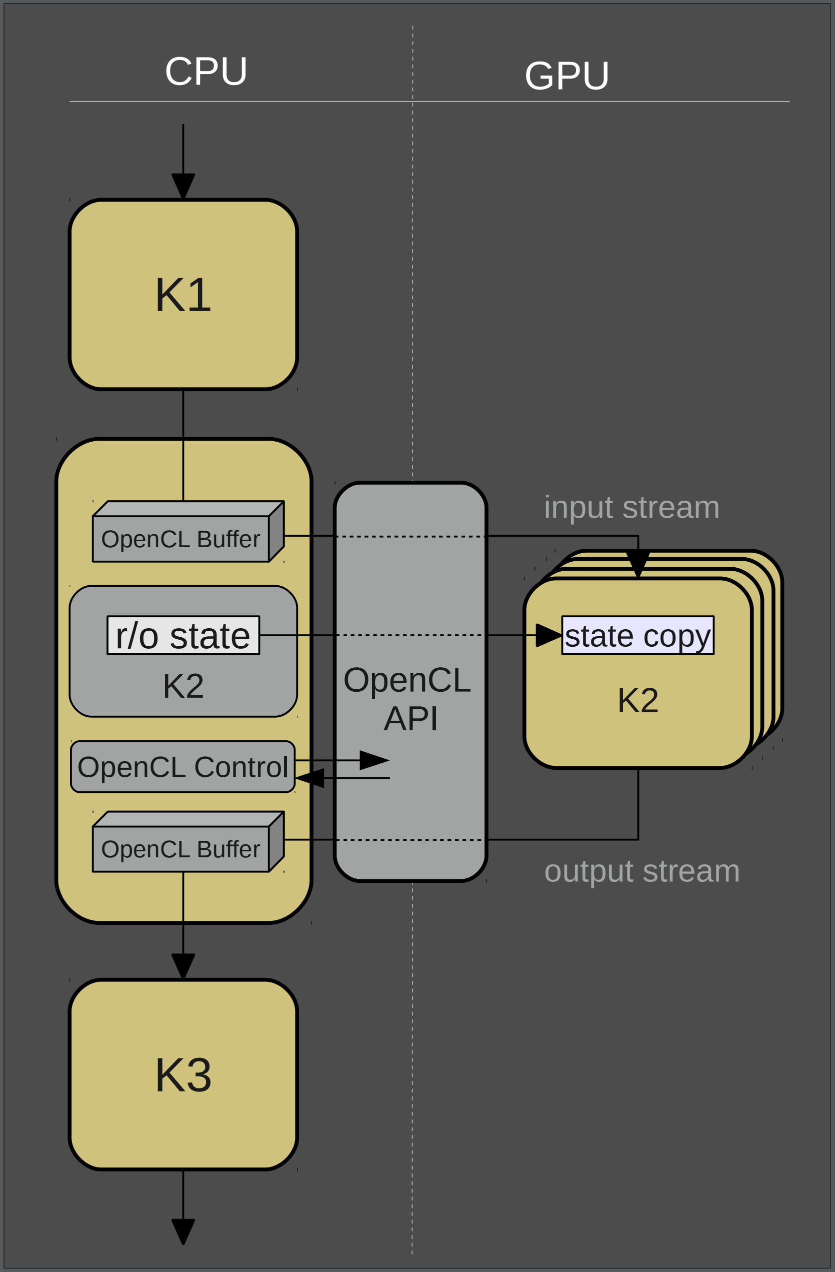
- Developed a fusion algorithm for SKIR
- Dynamic fusion shows performance benefits

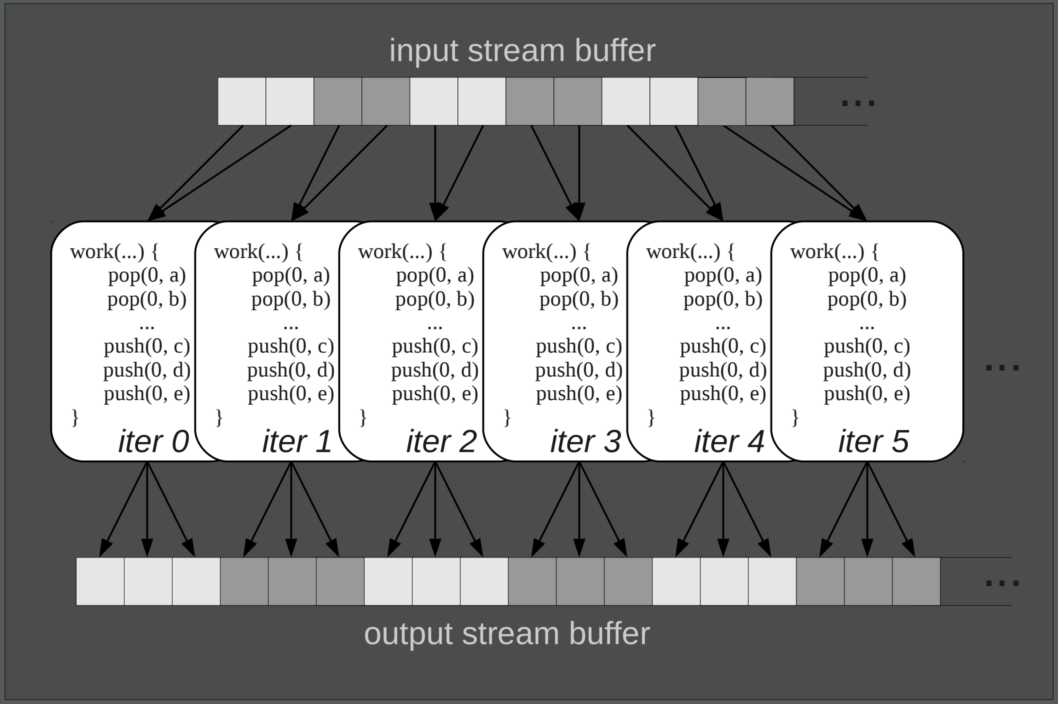
Compiling SKIR: Kernel Fission



- Kernel Fission is easy to implement for SKIR
- Automatic fission by SKIR runtime
- Manual fission by programmer or language
- One of many methods to exploit data parallelism

Compiling SKIR: OpenCL Backend





```
work(...) {
    pop(0, a)
    pop(0, b)
    ...
    push(0, c)
    push(0, d)
    push(0, e)
}
```

- Transparent execution on GPU via OpenCL
- Modified version of LLVM C backend to emit OpenCL kernels
- Any data parallel kernel with decidable state

Summary

- Optimized stream parallelism using LLVM
    - Dynamic compilation
    - Dynamic scheduling
- Performance
    - Good!
- Future work
    - Use for ongoing network & signal processing research
    - Better GPU support
    - Vectorization
- Open source soon

Contact:

fifield@colorado.edu
http://systems.cs.colorado.edu