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

---

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

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

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

void 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
}

int 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)
}
```

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( k = \text{skir.kernel work, arg} )</td>
<td>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 ).</td>
</tr>
<tr>
<td>( \text{skir.call } k, \text{ins, outs} )</td>
<td>Execute kernel ( k ) with the input streams ins and the output streams outs. ins and outs are arrays of stream objects.</td>
</tr>
<tr>
<td>( \text{skir.uncall } k )</td>
<td>Stop execution of ( k ) and remove it from the stream graph.</td>
</tr>
<tr>
<td>( \text{skir.wait } k )</td>
<td>Block until kernel ( k ) finishes execution.</td>
</tr>
<tr>
<td>( \text{skir.become } k )</td>
<td>Replace the currently executing kernel with ( k ). Must be called from within a kernel work function.</td>
</tr>
<tr>
<td>( s = \text{skir.stream size} )</td>
<td>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.</td>
</tr>
<tr>
<td>( \text{skir.push } idx, \text{data} )</td>
<td>Push ( \text{data} ) onto output stream ( idx ).</td>
</tr>
<tr>
<td>( \text{skir.pop } idx, \text{data} )</td>
<td>Pop an element from input stream ( idx ) and store the result into ( data ).</td>
</tr>
<tr>
<td>( \text{skir.peek } idx, \text{data, off} )</td>
<td>Read the stream element from input stream ( idx ) at offset ( off ) and store the result into ( data ).</td>
</tr>
</tbody>
</table>
```c
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
}

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)
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];
}
**SKIR as a compiler target: C++**

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

```
#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;
}
};
```

stream parallelism is embedded deep inside the application

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

```c
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();
    }
}
```
SKIR as a compiler target: JavaScript

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

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

Compiling SKIR: Overview

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

Performance

Portability
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;
}
```

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

1) Run kernel until data or buffer space runs out
2) Locate kernel causing blockage
3) Switch to blocking kernel

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...

...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.
Scheduling SKIR: Obtaining parallel execution with task stealing

1) Run kernel until data or buffer space runs out

2) Locate kernel causing blockage

3) Steal or spawn blocking kernel

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

High overhead for small kernels

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

int -> int filter adder
{
    work {
        push(pop()+pop());
    }
}
Compiling SKIR: Kernel Fusion

- 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

- 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