MiniLS → Streaming OpenMP → Work-Streaming

Albert Cohen
Léonard Gérard Cupertino Miranda Cédric Pasteur
Antoniu Pop Marc Pouzet
based on joint-work with Philippe Dumont and Marc Duranton

1 INRIA
2 ENS and Université Pierre et Marie Curie
3 ENS and Université Paris-Sud
4 Mines ParisTech
5 while at Philips then NXP Semiconductors

Synchon 2010 – December 3, 2010
1. Terminology

2. Streaming Data-Flow \( n \)-Synchronous Programming

3. Streaming Data-Flow \( n \)-Synchronous Extension of OpenMP

4. High-Level Parallelizing Compilation of MiniLS

5. Low-Level Work-Streaming Compilation

6. Perspectives
Data-Flow Computing

Kahn networks
Least fixpoint of a system of equations over continuous functions on infinite streams

→ Deterministic by definition

→ ≡ communicating processes over infinite FIFOs with blocking reads
Synchronous semantics

Static restriction of Kahn semantics to zero-buffer equations, with clock types

→ Communicating processes have the same logical clock
→ Represents a sequential circuit
→ Deadlock-free: causality analysis
→ Static, clock-directed generation of sequential code
Data-Flow \( n \)-Synchronous Computing

[Cf. Synchron 2010 presentations of Louis Mandel and Florence Plateau.]

Goals

- Facilitate the programming of complex signal-processing algorithms
- Expose slack for desynchronization purposes and distributed/parallel execution
- Retain safety and performance (static compilation) properties

\( n \)-synchronous semantics

Static restriction of Kahn semantics to \textit{bounded} buffer equations, with clock types

→ Communicating processes have synchronizable logical clock
  ▶ Involves a richer algebraic structure on clock types
  ▶ Synchronizability: \( \bowtie \)
  ▶ Precedence: \( \preceq \)

→ Represent a latency-insensitive circuit

→ Static, clock-directed code generation
  ▶ Translation to a \((0-)\)synchronous program
  ▶ Or direct code generation to imperative code with buffers.
Stream Computing

1st interpretation: data-parallel Kahn networks
Data-flow computing where internal state is exposed as explicit (external) delays

\[
y = f(x) \rightsquigarrow (y, m) = f_{\text{pure}}(x, \text{pre}(m))
\]
\[
t = g(z) \rightsquigarrow (t, m) = g_{\text{pure}}(z, \text{pre}^k(m))
\]

stateful = dependence distance 1
dependence distance \(k\)

2nd interpretation: sliding window operations
Data-flow computing where past stream history is a first-class citizen in the syntax

- Reduces the need for states/delays in many algorithmic patterns
- Eliminates the associated copy overhead
- Syntactic sugar
- Express multi-token (bursty) reactions and asymmetric rate-conversions of CSDF
  [Cf. Synchron 2011 presentation of Leonard Gérard.]
2. Streaming Data-Flow \( n \)-Synchronous Programming

1. Terminology

2. Streaming Data-Flow \( n \)-Synchronous Programming

3. Streaming Data-Flow \( n \)-Synchronous Extension of OpenMP

4. High-Level Parallelizing Compilation of MiniLS

5. Low-Level Work-Streaming Compilation

6. Perspectives
Directions of Work

Data-flow $n$-synchrony for deterministic, scalable parallelism

- Streaming data-flow
- Computation model
- Language extension
- Intermediate representation
- Back-end compilation and optimization
- Runtime for decoupled execution
3. Streaming Data-Flow \( n \)-Synchronous Extension of OpenMP

1. Terminology
2. Streaming Data-Flow \( n \)-Synchronous Programming
3. Streaming Data-Flow \( n \)-Synchronous Extension of OpenMP
4. High-Level Parallelizing Compilation of MiniLS
5. Low-Level Work-Streaming Compilation
6. Perspectives
Streaming Data-Flow Programming

input/output (list)
list ::= list, item
    | item
item ::= stream
    | stream >> window
    | stream << window
stream ::= var
    | array[expr]
expr ::= var
    | value

int s, Rwin[Rhorizon];
int Wwin[Whorizon];
input (s >> Rwin[burstR])
output (s << Wwin[burstW])

OpenMP 3.0 extensions [HiPEAC’11]

- Capture task-level, dynamic data flow
- Stream computing: sliding windows, rate conversion
  - Inspired by StreamIt
  - Richer abstractions for programming comfort
  - Avoids copy overhead and artificial introduction of state
- Working on $n$-synchronous semantics
- Target for the desynchronization of synchronous data-flow programs
New Clauses: input and output

```c
int x, z;
int X[horizon];
int A[3];

#pragma omp task input (x >> X[burst])
  // task code block
  // 2 < burst <= horizon
  ... = ... X[2];

// array of 3 streams
#pragma omp task input (A[0] >> z)
  // task code block
  ... = ... z ...;

// stream with window horizon 3
#pragma omp task input (A)
  // task code block

int y;
int B[42][2];

#pragma omp task output (y)
  // task code block
  y = ...

// stream of arrays of size 2 with window horizon 42
#pragma omp task input (y >> B[17][])
  // task code block
  for (int i=0; i<17; ++i) {
    ... B[i][0];
    ... B[i][1];
  }
```
Interaction With Data Parallelism

```c
#pragma omp parallel num_threads (2)
#pragma omp single
{
    for (i = 0; i < N; ++i) {
#pragma omp task firstprivate (i) output (o)
        o = work (i);
#pragma omp task input (o)
        more_work (o);
    }
}
```

```c
#pragma omp parallel num_threads (2)
{
    #pragma omp for
    for (i = 0; i < N; ++i) {
#pragma omp task firstprivate (i) output (o)
        o = work (i);
#pragma omp task input (o)
        more_work (o);
    }
}
```
Interaction With Data Parallelism

```c
#pragma omp parallel num_threads (2)
#pragma omp single
{
    for (p=head; p!=null; p=p->next) {
#pragma omp task firstprivate (p) output (o)
        o = work (p);
#pragma omp task input (o)
        more_work (o);
    }
}
```

```c
#pragma omp parallel num_threads (2)
#pragma omp single
{
    for (p=head; p!=null; p=p->next) {
#pragma omp task firstprivate (p) output (o) num_threads (2)
        o = work (p);
#pragma omp task input (o)
        more_work (o);
    }
}
```
Stateful Filters

```c
#pragma omp parallel
#pragma omp single
{
    int counter = 0;
    for (i = 0; i < N; ++i) {
        #pragma omp task input (counter) output (x, counter)
        {
            counter++;
            x = ... ;
        }
        #pragma omp task input (x)
        ... = ... x ...;
    }
}
```
for (i = 0; i < N; ++i) {
    if (condition_1 (i)) {
        #pragma omp task firstprivate (i) output (x)
        x = i ;
    }
    if (condition_2 (i)) {
        #pragma omp task firstprivate (i) input (x)
        y = x + i ;
    }
}

- Liveness?
- Boundeness?
- Is synchrony sufficient to solve the problem?
Delays

for (i = 0; i < M; ++i)
#pragma omp task output (x <= A[k])
    for (j = 0; j < k; ++j)
        A[j] = ...;

for (i = 0; i < N; ++i) {
#pragma omp task input (y) output (x)
    x = ... y ...;
#pragma omp task input (x) output (y)
    y = ... x ... ;
}

- Stateless alternative to pre
- But liveness and boundeness requires $n$-synchrony
Interaction With Barriers

for (i = 0; i < M; ++i)
#pragma omp task output (x \ll A[k])
   for (j = 0; j < k; ++j)
      A[j] = ...;

#pragma omp taskwait
// deadlock if internal stream buffer size < kM

   for (i = 0; i < N; ++i) {
#pragma omp task input (y) output (x)
      x = ... y ...;
#pragma omp task input (x) output (y)
      y = ... x ... ;
   }

- Critically depends on \( n \)-synchrony!
More Combinations

- Nesting of parallel regions, tasks and work-sharing constructs
- Dynamic creation of tasks in (sequential or parallel) loops
- Variable burst size (with fixed horizon)
#pragma omp parallel
#pragma omp single
{
    float x, STR[2*(int)(log(N))];

    // Generate some input data
    for(i = 0; i < 2 * N; ++i)
#pragma omp task output (STR[0] << x)
    x = (i % 8) ? 0.0 : 1.0;

    // Reorder
    for(j = 0; j < log(N)-1; ++j) {
        int chunks = 1 << j;
        int size = 1 << (log(N) -j + 1);
#pragma omp task
        {
            float X[size], Y[size];
            float *w = compute_coefficients (size/2);

            for (i = 0; i < chunks; ++i) {
#pragma omp task input (STR[j] >> X[size]) output (STR[j+1] << Y[size]) shared (w)
                for (k = 0; k < size/2; k += 2) {
                    float t_r = X[size/2+k]*w[k] - X[size/2+k+1]*w[k+1];
                    float t_i = X[size/2+k]*w[k+1] + X[size/2+k+1]*w[k];
                    Y[k] = X[k] + t_r;
                    Y[k+1] = X[k+1] + t_i;
                    Y[(k+size)/2+1] = X[k+2];
                    Y[(k+size)/2+2] = X[k+3];
                }
            }
        }
    }

    // DFT
    for(j = 1; j <= log(N); ++j) {
        int chunks = 1 << (log(N) - j);
        int size = 1 << (j + 1);
#pragma omp task
        {
            float X[size], Y[size];
            float *w = compute_coefficients (size/2);

            for (i = 0; i < chunks; ++i) {
#pragma omp task input (STR[j+log(N)-2] >> X[size]) output (STR[j+log(N)-1] << Y[size]) shared (w)
                for (k = 0; k < size/2; k += 2) {
                    float t_r = X[size/2+k]*w[k] - X[size/2+k+1]*w[k+1];
                    float t_i = X[size/2+k]*w[k+1] + X[size/2+k+1]*w[k];
                    Y[k] = X[k] + t_r;
                    Y[k+1] = X[k+1] - t_r;
                    Y[size/2+k] = X[k+2] - t_i;
                    Y[size/2+k+1] = X[k+3] - t_i;
                }
            }
        }
    }

    // Output the results
    for(i = 0; i < 2 * N; ++i)
#pragma omp task input (STR[2*log(N)-1] >> x)
    printf ("%f\t", x);
}
Example: FFT

```
// DFT
for(j = 1; j <= log(N); ++j) {
    int chunks = 1 << (log(N) - j);
    int size = 1 << (j + 1);
    #pragma omp task
    {
        float X[size], Y[size];
        float *w = compute_coefficients (size/2);
        for (i = 0; i < chunks; ++i) {
            #pragma omp task input (STR[j+log(N)-2] >> X[size]) output (STR[j+log(N)-1] << Y[size]) shared (w)
            for (k = 0; k < size/2; k += 2) {
                float t_r = X[size/2+k]*w[k] - X[size/2+k+1]*w[k+1];
                float t_i = X[size/2+k]*w[k+1] + X[size/2+k+1]*w[k];
                Y[k] = X[k] + t_r;
                Y[k+1] = X[k+1] + t_i;
                Y[size/2+k] = X[k] - t_r;
                Y[size/2+k+1] = X[k+1] - t_i;
            }
        }
    }
}

// Output the results
for(i = 0; i < 2 * N; ++i)
    printf("\%f\t", x);
```

## Reorder stages

```
// Reorder
for(j = 0; j < log(N)-1; ++j) {
    int chunks = 1 << j;
    int size = 1 << (log(N) - j + 1);
    #pragma omp task
    {
        float X[size];
        float Y[size];
        for (i = 0; i < chunks; ++i) {
            #pragma omp task input (STR[j] >> X[size]) output (STR[j+1] << Y[size])
            for (k = 0; k < size; k+=4) {
                Y[k/2] = X[k];
                Y[k/2+1] = X[k+1];
                Y[(k+size)/2+1] = X[k+2];
                Y[(k+size)/2+2] = X[k+3];
            }
        }
    }
}
```
4. High-Level Parallelizing Compilation of MiniLS

1. Terminology

2. Streaming Data-Flow \( n \)-Synchronous Programming

3. Streaming Data-Flow \( n \)-Synchronous Extension of OpenMP

4. High-Level Parallelizing Compilation of MiniLS

5. Low-Level Work-Streaming Compilation

6. Perspectives
Aim for SPMD: Simplest Possible Modular Design

async node pipe (i1, i2) outputs (o1, o2)
let
  o1 = a(i1);
  async x = f(i1, o1);
  o = g(x, i1 fby i2);
  o2 = b(o)
let

let (r1, r2) = pipe(42, 17)

↓

main() {
#pragma omp parallel
#pragma omp single
{
  pipe.reset(42);
  while (true) {
    int r1, r2;
    #pragma omp task output (r1, r2)
    pipe.astep(42, 17, &r1, &r2);
  }
}
} // end main

obc a {
  method reset () ...
  method step (i1) ...
}

...

obc pipe {
  mem int i2;

  method reset (int i1) {
    #pragma omp task firstprivate (int i1) output (int i2)
    i2 = i1; return i2;
  }

  method step (int i1, int i2) {
    ...
  }

  method astep (int i1, int i2, int *o1_p, int *o2_p) {
    int x, o;
    #pragma omp task firstprivate (i1) output (o1)
    o1 = a.step(i1); // a.step: { o1 = a(i1); return o1; }
    #pragma omp task firstprivate (i1) input (o1) output (x)
    x = f.step(i1, o1); // f.step: { x = f(i1, o1); return x; }
    #pragma omp task input (i2, x) output (o2)
    {
      o = g.step(x, i2); // g.step: { o = g(x, i2); return o; }
      o2 = b.step(o); // b.step: { o2 = b(o); return o2; }
    } // end step
  }
}
5. **Low-Level Work-Streaming Compilation**

1. Terminology
2. Streaming Data-Flow $n$-Synchronous Programming
3. Streaming Data-Flow $n$-Synchronous Extension of OpenMP
4. High-Level Parallelizing Compilation of MiniLS
5. **Low-Level Work-Streaming Compilation**
6. Perspectives
Intermediate Representation for Stream Computing

Question
Scalable and efficient compilation of data-flow streaming programs?

The three goals of Erbium [CASES’10]

1. Express deterministic multi-producer multi-consumer, task- and data-parallel computations
2. Eliminate runtime overhead, amortize hardware synchronization costs
3. Nothing to hide to the compiler
   - Decouple synchronization, communication, access to local buffers
   - Support aggressive scalar, loop and interprocedural optimization
**Erbium Intermediate Representation and Runtime**

- **record**: multi-producer, multi-consumer stream
- **view**: randomly addressable sliding window, read or write side
- **commit()**/**update()**: pressure
- **release()**/**stall()**: back-pressure
- **receive()**: one-sided, asynchronous communication
- Deterministic initialization protocol and garbage collection

**Lightweight runtime**

- Wait-free, consensus-free implementation: no hardware atomic instruction, no fence
- \( \approx 10 \) cycles per streaming communication cycle
- Compatible with a work-stealing scheduler
Enables Task-Level Optimization

Important optimizations enabled by Erbium

- Conversion to persistent streaming processes
  - Scalable parallel execution of data-flow tasks with streaming constructs
- Task data-parallelization
  - Parallel iteration of independent activations of a task
  - Thread-level and vector parallelism
- Dynamic task coarsening
  - Sequential iteration of a task to hide latency
- Synchronization optimization
  - Elimination of redundant \texttt{update()}s/\texttt{stall()}s.

Some optimizations may be better handled at a higher semantical level

- Task fusion and scheduling
  - Static code generation, clock-directed
- Static task coarsening
  - Loop nest transformation analog: strip-mining
Work-Streaming Code Generation

Example: data-parallel task

```c
float x, y;
#pragma omp parallel for
for (...) {
    #pragma omp task input(x) output(y)
    y = f(x);
}
```

↓ Work-streaming compilation and runtime ↓

```c
record float *s_x, *s_y;
init(s_x, ...);
init(s_y, ...);
allocate(s_x, ...);
allocate(s_y, ...);
for (i=0; i<nb_workers; i++)
    run persistent_task();
```

```c
while(true) { // Code of a persistent streaming task
    int beg, end, beg_s, end_s;
    ask_for_work(s_x, &beg, &end); // work-stealing (blocking)
    for (beg_s=beg; beg_s<=end; beg_s+=AGGREGATE) {
        end_s = MIN(beg_s+AGGREGATE, end);
        stall(s_y, end_s); // blocking
        receive(s_x, beg_s, end_s); // non-blocking
        update(s_x, end_s); // blocking
        for (i=beg_s; i<end_s; i+=4)
            s_y[i..i+3] = f_v4f_clone(s_x[i..i+3]);
        for (max(0, i-4); i<end_s; i++)
            s_y[i] = f(s_x[i]);
        commit(s_y, end_s); // non-blocking
    }
}
```
Application to FFT

Best configuration for each FFT size

4-socket Opteron – 16 cores
Combination of Task-Level and Low-Level Optimizations

Example: fmradio (from GNUradio)

Platform – cores | Seq. –03 | Par. –02 | Par. –03 | Par. –03 vs. Par. –02
--- | --- | --- | --- | ---
Xeon – 24 cores | 1.14 | 10.1 | 12.6 | 1.25
Opteron – 16 cores | 1.52 | 9.51 | 14.6 | 1.54
6. Perspectives

1. Terminology

2. Streaming Data-Flow $n$-Synchronous Programming

3. Streaming Data-Flow $n$-Synchronous Extension of OpenMP

4. High-Level Parallelizing Compilation of MiniLS

5. Low-Level Work-Streaming Compilation

6. Perspectives
What’s Next?

- Definition of the $n$-synchronous semantics of the streaming extension
- Contributing to the OpenMP language specification
- Parallelizing compilation of $n$-synchronous Kahn networks
- Scalable parallelization with burst-synchronous Kahn networks
- Implementation of the work-streaming compilation algorithm
- Task-level optimization (coupled with polyhedral compilation)