A Stream-Computing Extension to OpenMP

Antoniu Pop and Albert Cohen

antoniu.pop@mines-paristech.fr, albert.cohen@inria.fr

INRIA — MINES ParisTech

Contribution: enable expressing and efficiently exploiting pipeline parallelism in OpenMP programs

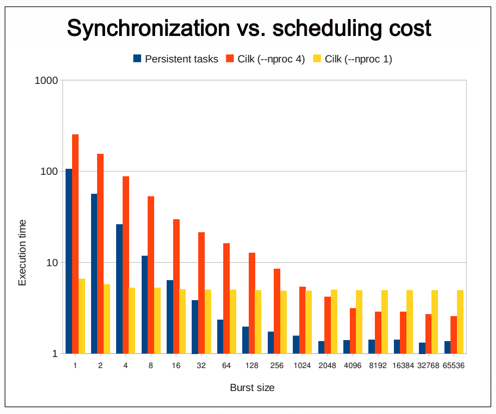
Motivation: OpenMP needs a strategy for programming and exploiting current architectures

- Data-parallelism is hard to exploit on complex memory hierarchies
- Pipelining has a structuring effect on communication, which improves cache behaviour
- Scheduling fine-grained tasks is often less efficient than synchronizing persistent tasks

We compare the optimized fine-grained task scheduling in Cilk with persistent streaming tasks implemented with Erbium.

The synchronization algorithm is lock-free, uses no hardware atomic operations or fences and is optimized for minimizing cache traffic.

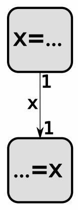
We use the exploration kernel with only one multiply-add per transaction to show the overhead incurred in the runtime.



Streaming constructs

Simple pipeline:

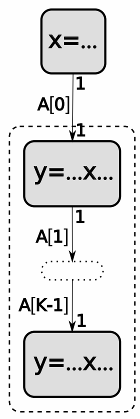
```
#pragma omp parallel
#pragma omp single
  for (i = 0; i < N; ++i)
   {
#pragma omp task output (x)
       x = ... ;
#pragma omp task input (x)
       ... = ... x ...;
   }
```



- Views can be implicit: **x** represents both the stream and the view
- The connector operator and the view can be omitted in the streaming clauses
- Burst and peek values are implicitly 1 for scalars

Dynamic pipeline of filters:

```
int A[K];
#pragma omp parallel
#pragma omp single
  for (i = 0; i < N; ++i) {
#pragma omp task output (A[0] << x)
       x = ... ;
   }

  for (j = 0; j < K-1; ++j)
#pragma omp task
   {
      for (i = 0; i < N; ++i)
       {
#pragma omp task input (A[j] >> x)\
                 output (A[j+1] << y)
            y = ... x ...;
       }
   }
```



- Building a dynamic pipeline requires an array of streams: **A**
- This code builds a pipeline of K+1 tasks connected by K streams
- The non-streaming OpenMP task cnstruct is used to create multiple filter instances
- The views **x** and **y** are connected to the streams **A[...]**

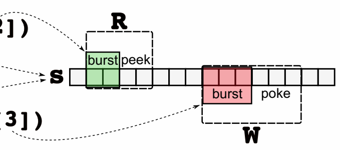
Language extension:

- add input and output clauses for OpenMP3.0 task constructs

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

Semantics of clauses: connecting views to streams

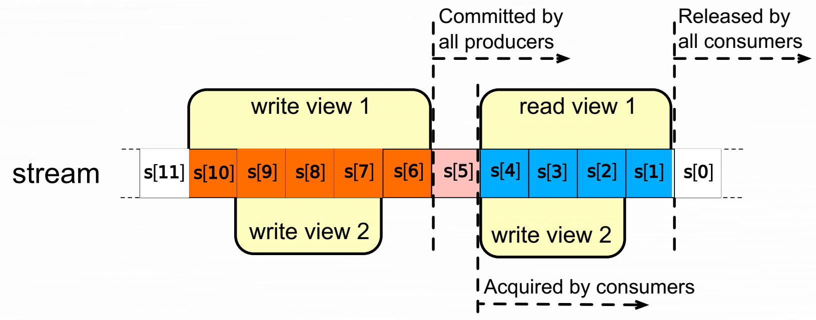
```
int s, R[4], W[6];

input (s >> R[2])

output (s << W[3])
```



- make pipelined tasks persistent
  - preserve the semantics
  - improve performance

Streaming runtime: Erbium

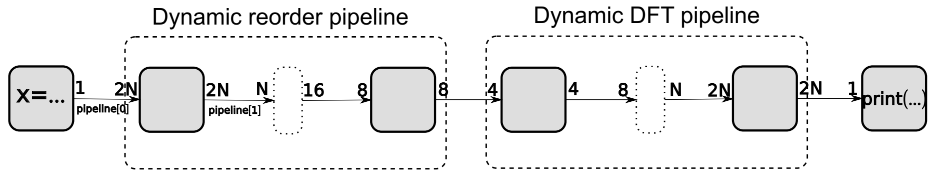
- Multi-Producer - Multi-Consumer streams
- Connect multiple read/write views to a stream using the >> connector



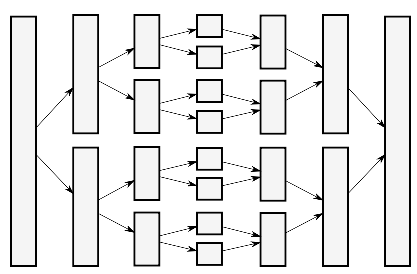
Detailed example: FFT streamization

```
#pragma omp parallel
{
#pragma omp single
{
  float x;
  float pipeline[2*(int)(log(N))];

  // Generate some input data (or read from a file)
  for(i = 0; i < 2 * N; ++i) {
#pragma omp task output (pipeline[0] << x)
    x = (i % 8) ? 0.0 : 1.0;
  }

  // Reorder stages
  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 (pipeline[j] >> X[size])\
                 output (pipeline[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];
          }
        }
      }
    }
  }

  // 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[size/2];

      // ... compute the coefficients in w[]

      for (i = 0; i < chunks; ++i) {

#pragma omp task input (pipeline[j+log(N)-2] >> X[size]) \
                 output (pipeline[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)
#pragma omp task input (pipeline[2*log(N)-1] >> x, stdout)
                 output (stdout)
    printf ("%f\t", x);
}}
```

Taskgraph of the streamized FFT

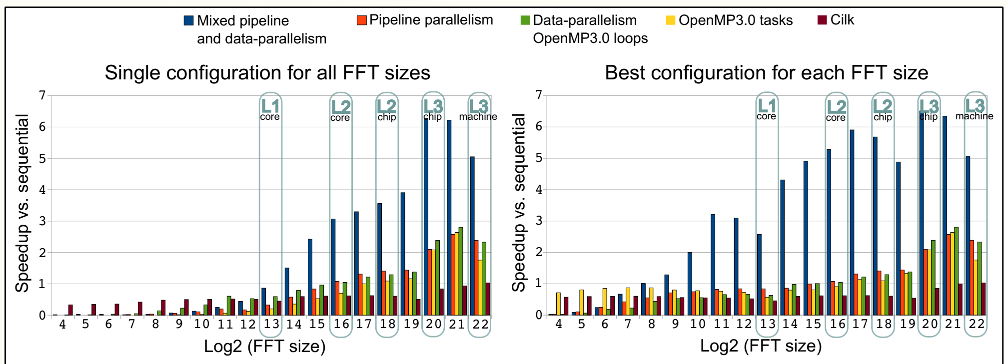


FFT data-flow graph



- Pipelined FFT allows wavefront parallelization
- Data-parallelism is available in each stage (vertical slice)
- Granularity can be controlled by the number of times the data is split before applying the sequential algorithm

Target 1:
- 4-socket AMD quad-core Opteron 8380 (Shanghai) with 16 cores at 2.5GHz
- 64GB of memory, 64KB per core L1, 512KB per core L2, 6MB per chip L3



Target 2:
- 4-socket Intel hexa-core Xeon E7450 (Dunnington) with 24 cores at 2.4GHz
- 64GB of memory, 32KB per core L1, 3MB per two cores L2, 12MB per chip L3