OpenMP and Work-Streaming Compilation in GCC

Antoniu Pop

Centre de recherche en informatique, MINES ParisTech

GROW'11 — 3 April 2011, Chamonix, France

MINES ParisTech

Introduction
No surprise there is a memory wall issue

Possible solution: stream-computing

- Favors local, on-chip communication
- Hides memory latency
- Transparent aggregation of communications

GCC can benefit from a streaming-enabled OpenMP implementation

The current OpenMP compilation can be improved

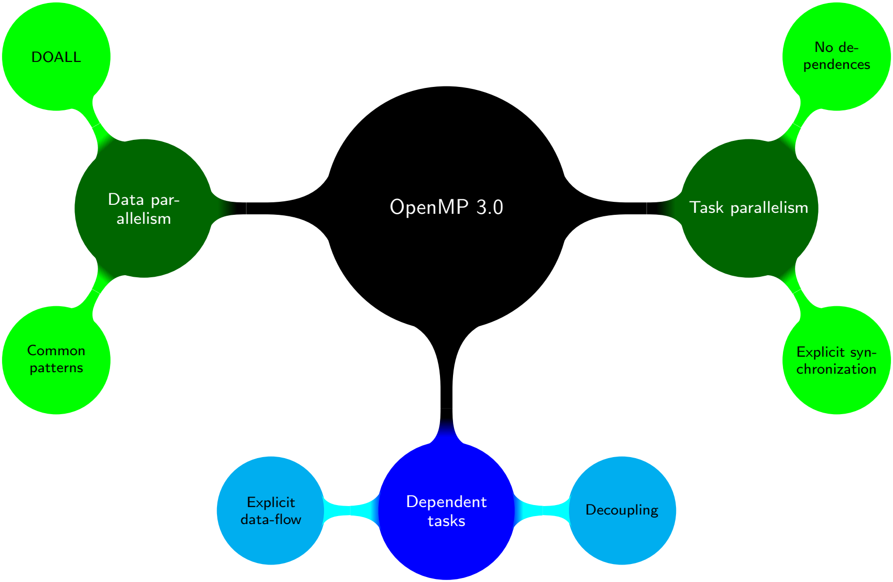
Outline

1. OpenMP Background
2. Streaming in OpenMP
3. OpenMP Compilation in GCC
4. Work-Streaming Compilation
5. Improving OpenMP Compilation

1. OpenMP Background

1. OpenMP Background
2. Streaming in OpenMP
3. OpenMP Compilation in GCC
4. Work-Streaming Compilation
5. Improving OpenMP Compilation

Bird's Eye View of OpenMP



Crash Course on OpenMP 3.0 Tasks (programming model)

Current semantics: similar to coroutines

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

```
#pragma omp parallel num_threads (2)
{
#pragma omp for
  for (i = 0; i < N; ++i)
#pragma omp task firstprivate (i)
    {
      work (i);
    }
}
```

- Single-entry single-exit regions (no branching in or out)
- Sharing clauses: private, firstprivate, shared
- Synchronization: taskwait, locks
- **Communication restricted to shared memory and manual synchronization**
  - Conservative over-synchronization
  - Error-prone
  - Poor memory locality

2. Streaming in OpenMP

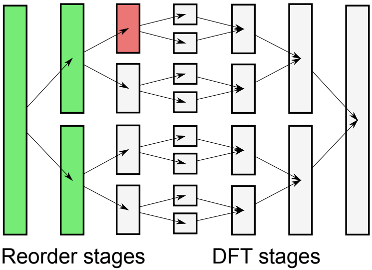
1. OpenMP Background
2. **Streaming in OpenMP**
3. OpenMP Compilation in GCC
4. Work-Streaming Compilation
5. Improving OpenMP Compilation

Motivation for Streaming

Sequential FFT implementation

```
float A[2 * N];
for(i = 0; i < 2 * N; ++i)
  A[i] = (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);

  for (i = 0; i < chunks; ++i)
    for (k = 0; k < size; k+=4)
      reorder (A[i*size .. (i+1)*size-1]);
}
```

```
// DFT
for(j = 1; j <= log(N); ++j) {
  int chunks = 1 << (log(N) - j);
  int size = 1 << (j + 1);

  for (i = 0; i < chunks; ++i)
    for (k = 0; k < size/2; k += 2)
      compute_DFT (A[i*size .. (i+1)*size-1]);
}

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

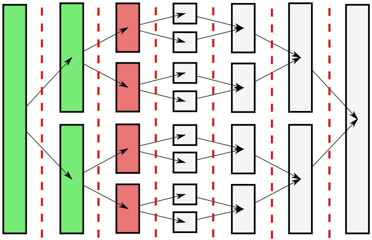


Example: FFT Data Parallelization

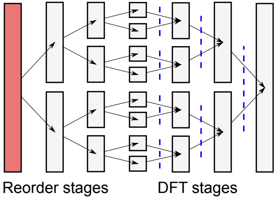
OpenMP parallel loop implementation

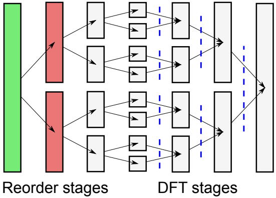
```
float A[2 * N];
for(i = 0; i < 2 * N; ++i)
  A[i] = (i % 8) ? 0.0 : 1.0;

// Reorder
for(j = 0; j < log(N)-1; ++j)
{
  int chunks = 1 << j;
  int size = 1 << (log(N) -j + 1);

#pragma omp parallel for
  for (i = 0; i < chunks; ++i)
    for (k = 0; k < size; k+=4)
      reorder (A[i*size .. (i+1)*size-1]);
}
```

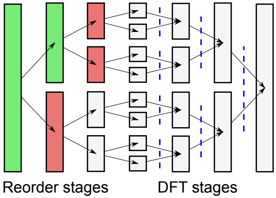
```
// DFT
for(j = 1; j <= log(N); ++j) {
  int chunks = 1 << (log(N) - j);
  int size = 1 << (j + 1);

#pragma omp parallel for
  for (i = 0; i < chunks; ++i)
    for (k = 0; k < size/2; k += 2)
      compute_DFT (A[i*size .. (i+1)*size-1]);
}

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

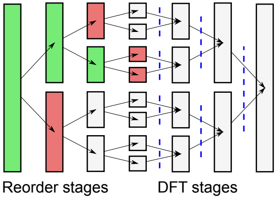


Example: FFT Task Parallelization

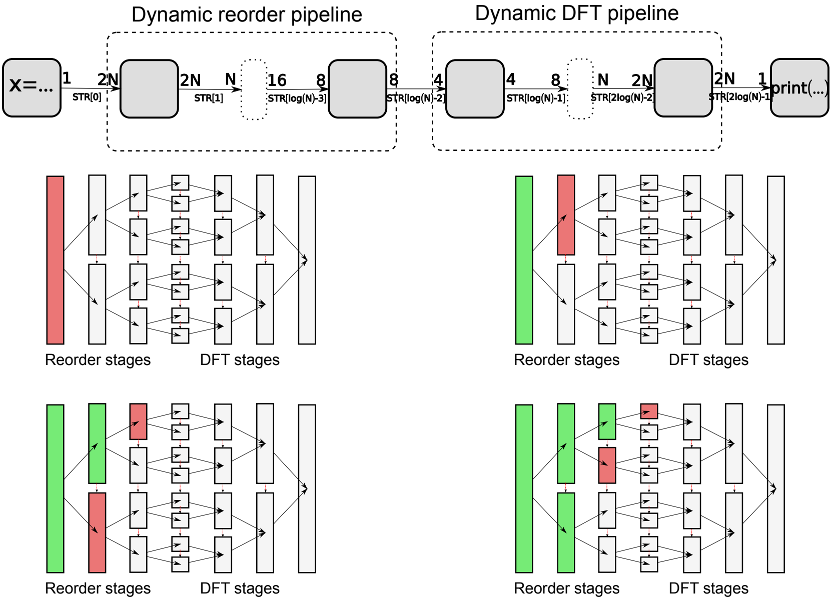




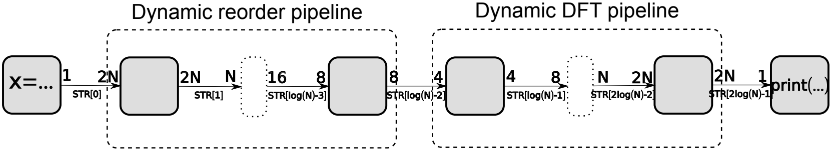


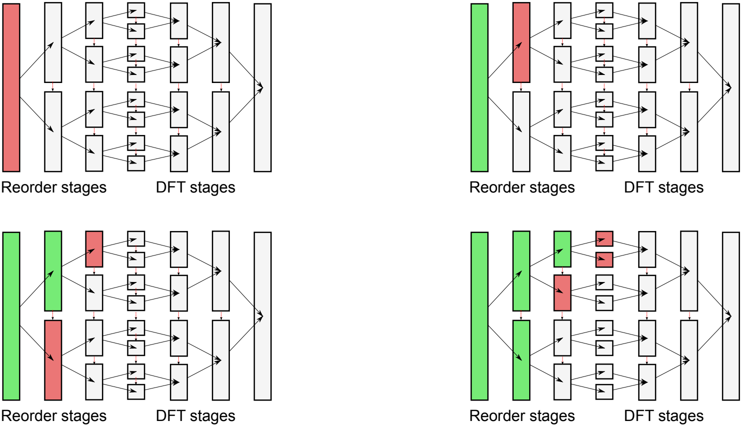


Example: FFT Pipeline Parallelization

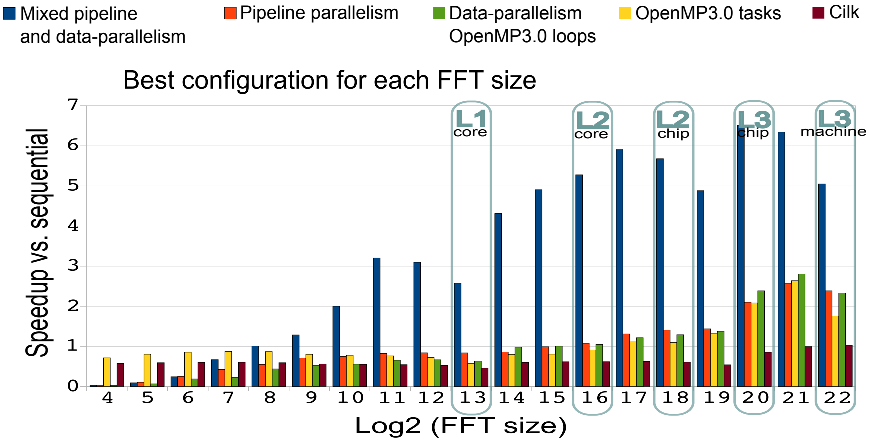


Example: FFT Streamization (pipeline and data-parallelism)





Evaluation of FFT Parallelization Techniques

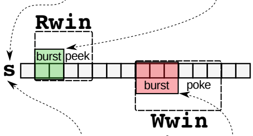


4-socket Opteron – 16 cores

OpenMP Extension for Stream-Computing: Syntax

```
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])
```

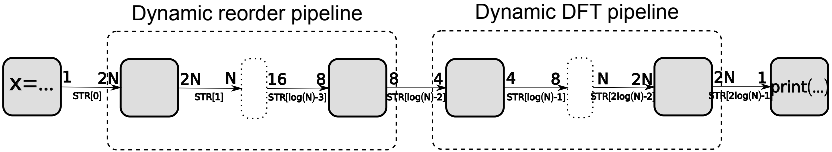



```
output (s << Wwin[burstW])
```

```
int S[K];        // Array of streams
int X[horizon]; // View

#pragma omp task output (S[0] << X[burst])
  // task code block
  // burst <= horizon
  for (int i = 0; i < burst; ++i)
    X[i] = ...;

#pragma omp task input (S[0] >> X[burst])
  // task code block
  // burst <= horizon
  for (int i = 0; i < horizon; ++i)
    ... = ... X[i];
```

```
int A[5];    // Stream of arrays

#pragma omp task output (A)
  // task code block
  // Each element is an array
  for (int i = 0; i < 5; ++i)
    A[i] = ...

#pragma omp task input (A)
  // task code block
  for (int i = 0; i < 5; ++i)
    ... = ... A[i];
```

Streamized FFT Implementation with the OpenMP Extension



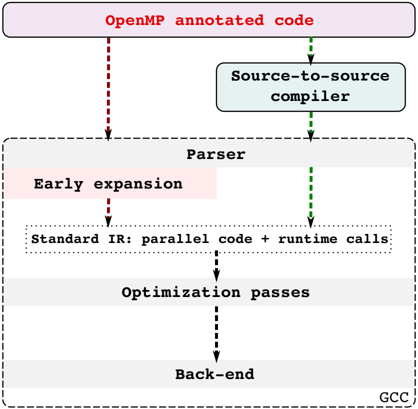

```c
float x, STR[2*(int)(log(N))];

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);
  float X[size], 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[0..size-1] = reorder (X[0..size-1]);
    }
}
```

```c
// DFT
for(j = 1; j <= log(N); ++j) {
  int chunks = 1 << (log(N) - j);
  int size = 1 << (j + 1);
  float X[size], Y[size];

  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])
    for (k = 0; k < size/2; k += 2)
    {
      Y[0..size-1] = compute_DFT (X[0..size-1]);
    }
}

for(i = 0; i < 2 * N; ++i)
#pragma omp task input(STR[2*log(N)-1] >> x)\
                 input (stdout) output (stdout)
  printf ("%f\t", x);
```

3. OpenMP Compilation in GCC

1. OpenMP Background
2. Streaming in OpenMP
3. **OpenMP Compilation in GCC**
4. Work-Streaming Compilation
5. Improving OpenMP Compilation

OpenMP Compilation Flow



Code generation

- Direct translation of annotations to runtime calls
- Minimalistic static analysis

OpenMP Task Compilation in GCC

```
for (i = 0; i < N; ++i) {
  if (condition ()) {
#pragma omp task firstprivate (i)
    work (i);
  }
}
```

```
void task_function (&params) {
  i = params->i;
  work (i);
}

for (i = 0; i < N; ++i) {
  if (condition ()) {
    params.i = i;
    GOMP_task (task_function, &params, ...);
  }
}
```

Outer context ("main program")

- Outline the task body
- Add marshalling code for sharing clauses
- Insert runtime call to spawn the task

Inner context ("task body")

- Add unmarshalling code

OpenMP Task Execution Model

Outer context: `GOMP_task` call

- enqueue the work function pointer and the parameter structure on the scheduler
- execute the work function

Scheduler

- Dynamic scheduling
- Scheduler queue bound to the outer context (nested tasks)
- No order can be assumed
- Work function considered side-effect free

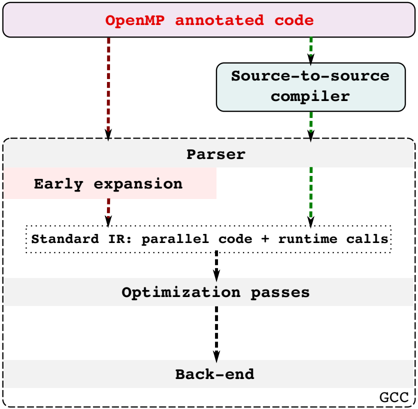
Synchronization

- Locks
- Barriers: wait until all outstanding tasks complete
- Taskwait: wait until all outstanding tasks issued by the current context complete

4. Work-Streaming Compilation

1. OpenMP Background
2. Streaming in OpenMP
3. OpenMP Compilation in GCC
4. Work-Streaming Compilation
5. Improving OpenMP Compilation

Work-Streaming Compilation Flow



- No additional pass
- Integrated in the OpenMP compilation flow

Work-Streaming Compilation

Example of streaming task

```
float x, y;
for (i = 0; i < N; ++i) {
  // Do non-streaming work
  if (condition ()) {
#pragma omp task input(x) output(y)
    y = f (x);
  }
}
```

Direct translation

- Outline the task body
- Insert runtime calls

Task-Level Optimizations

- Conversion to persistent streaming processes
- Data and work aggregation
- Data-parallelization

Work-Streaming Code Generation (base case)

Example: streaming task

```
float x, y;
for (i = 0; i < N; ++i) {
  // Do non-streaming work
  if (condition ()) {
#pragma omp task input(x) output(y)
    y = f (x);
  }
}
```

↓ Work-streaming compilation and runtime ↓

```
GOMP_stream_id id_x, id_y;
for (i = 0; i < N; ++i) {
  // Do non-streaming work
  if (condition ()) {
    GOMP_activate_stream_task
      (stream_task_wf, id_x, id_y);
  }
}
```

```
void stream_task_wf (&params) {
  GOMP_stream s_x = params->x, s_y = params->y;
  float *view_x, *view_y;
  int current;

  while (get_activation (&current)) {
    view_y = stall (s_y, current); // blocking
    view_x = update (s_x, current); // blocking

    *view_y = f (*view_x);

    commit (s_y, current); // non-blocking
    release (s_x, current); // non-blocking
  }
}
```

Task-Level Optimizations

Conversion to persistent streaming processes

- Reduce scheduling overhead for fine-grained tasks
- Rely on efficient synchronization of stream accesses

Aggregation

- Sequential iteration of task activations
- Reduce runtime overhead
- Avoid cache misses from false sharing
- Enable thread-level vectorization

Data-parallel tasks

- Stateless tasks are (by definition) data-parallel
- Multiple threads execute on separate iteration blocks

Work-Streaming Code Generation (optimized case)

```
GOMP_stream_id id_x, id_y;
for (i = 0; i < N; ++i) {
  // Do non-streaming work
  if (condition ()) {
    GOMP_activate_stream_task
      (stream_task_wf, id_x, id_y);
  }
}
```

```
void stream_task_wf (&params) {
  GOMP_stream s_x = params->x, s_y = params->y;
  float *view_x, *view_y;
  int beg, end, beg_s, end_s;

  while (get_activation_range (&beg, &end)) {
    for (beg_s=beg; beg_s<=end; beg_s += AGGREGATE) {
      end_s = MIN (beg_s + AGGREGATE, end);
      view_y = stall (s_y, end_s); // blocking
      view_x = update (s_x, end_s); // blocking

      // Automatic vectorized version
      for (i=0; i<end_s-beg_s; i+=4)
        view_y[i..i+3] = f_v4f_clone (view_x[i..i+3]);

      // Fall-back version
      for (MAX (0, i-4); i<end_s-beg_s; i++)
        view_y[i] = f (view_x[i]);

      commit (s_y, end_s); // non-blocking
      release (s_x, end_s); // non-blocking
    }
  }
}
```

- Views directly access stream buffers: no unwarranted memory copy
- GCC vectorization automatically handles the regular loop

Work-Streaming

Task activation

- Compressed if-conversion: only true instances matter
  - Avoid spurious activations
  - Do not activate a task just to decide there's nothing to do
- Control-driven data-flow computing
  - Allows deterministic merge of multi-task output to a stream
  - Schedule data based on control-flow
  - Simplified schedule of task activations

Work Aggregation

- Liveness guarantees: task activation availability ensures liveness
- Fairness requires additional runtime support for work-stealing
  - Preempt work that has already been acquired by a concurrent thread
  - Steal-back from a thread that bites more than it can (or should) chew

Data Aggregation

- Liveness is an issue
  - Runtime needs to detect the presence of strongly connected components in the taskgraph
  - Default fall-back to no data aggregation within cycles

Runtime Support Implemented in libGOMP

Stream synchronization

- Efficient synchronization algorithm
  - Aims at minimizing cache traffic
  - Lock- and atomic operation- free
- Deterministic data schedule avoids contention

Dynamic taskgraph

- Flow dependence relations
- Static taskgraph possible
  - Trivial to build
  - Over-approximation of the dynamic graph
- Needed for enabling data aggregation and deadlock detection

Deadlocks and Deadlock-Freedom Conditions

Formal model (on-going work)

- Deadlock-freedom proved for stream-causal programs
- Spurious deadlock-freedom for strict OpenMP-semantics-compliant programs
- Static over-approximation of possible deadlock condition

Runtime deadlock-detection algorithm

- Enabled for tasks that meet the static over-approximation (with debug flag)
- Use stalling time
  - Explore the dynamic taskgraph
  - Find strongly connected components
  - Evaluate precise data-dependence relations
- Debugging support
  - Find precise information on deadlock conditions
  - Identify involved tasks

5. Improving OpenMP Compilation

1 OpenMP Background

2 Streaming in OpenMP

3 OpenMP Compilation in GCC

4 Work-Streaming Compilation

5 Improving OpenMP Compilation

Example: a (very) simple OpenMP program

```
int main () {
  int *a = ... ;

#pragma omp parallel for shared (a) schedule (static)
    for (i = 0; i < N; ++i)
      {
        a[i] = foo (...);
      }

  for (j = 0; j < N; ++j)
    ... = a[j]
}
```

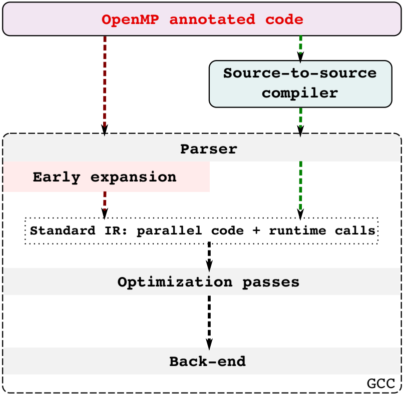
- Static schedule: generates the simplest code to analyze

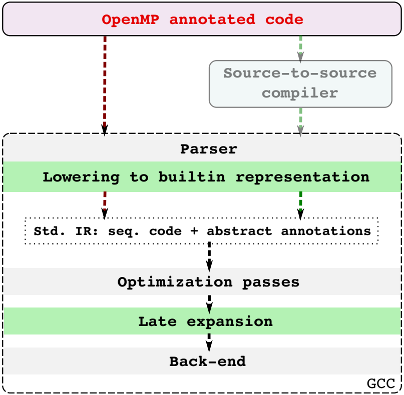
Early expansion of OpenMP annotations in GCC

```
void main_omp_fn_0 (struct omp_data_s * omp_data_i) {
  n_th = omp_get_num_threads();
  th_id = omp_get_thread_num();
  // compute lower and upper bounds from n_th and th_id

  for (i = lower; i < upper; ++i) {
    omp_data_i->a[i] = foo (...);
  }
}

int main () {
  int *a = ... ;

  omp_data_o.a = a;
  GOMP_parallel_start (main_omp_fn_0, &omp_data_o, 0);
  main_omp_fn_0 (&omp_data_o);
  GOMP_parallel_end ();
  a = omp_data_o.a;

  for (j = 0; j < N; ++j)
    ... = a[j]
}
```

- Few optimization opportunities for OpenMP-annotated code after expansion, even for simple and crucial sequential optimizations
- Opportunities for optimizing the exploitation of parallelism are lost

Avoiding the early expansion pass





Translation to Builtin Representation

```
int main () {
  int *a = ... ;

#pragma omp parallel for shared (a) schedule (static)
    for (i = 0; i < N; ++i)
      a[i] = foo (...);

  for (j = 0; j < N; ++j)
    ... = a[j]
}
```

↓ Lowering to builtin representation ↓

```
int main () {
  int *a = ... ;

  if (__property_parallel () && __property_for ()
       && __property_shared (&a) && __property_schedule (static))
  {
    for (i = 0; i < N; ++i)
      a[i] = foo (...);
  }

  for (j = 0; j < N; ++j)
    ... = a[j]
}
```

OpenMP Late Expansion

What do we stand to gain ?

- Enables serial optimizations for free
  - Compiler can analyze the code
  - Existing optimizations apply as they are
  - Special care is needed to inhibit destructive optimizations
- More statical analysis information available at later stages of the compilation flow
  - Data-dependences
  - SSA representation
- Optimization of parallel code
- Use annotation information in optimization passes

Example: PRE

```
  x = 2;
  y = 3;

#pragma omp parallel shared (a) firstprivate (x,y)
  {
#pragma omp single
    {
      for (i = 0; i < N; ++i)
#pragma omp task shared (a)
        a = x + y;
    }
  }
  // use a
```

Should (roughly) be optimized down to:

```
  // ... nothing

  // use 5
```

Example: PRE (continued)

Out of reach with early expansion due to marshalling and outlining

```
// main
  .omp_data_o.6.y = 3; // y
  .omp_data_o.6.x = 2; // x
  .omp_data_o.6.a = a;
  GOMP_parallel_start (main._omp_fn.0, &.omp_data_o.6, ...);
  main._omp_fn.0 (&.omp_data_o.6);
  GOMP_parallel_end ();
  a = .omp_data_o.6.a;

// main._omp_fn.0 (struct .omp_data_s.2 * .omp_data_i)
  .omp_data_o.5.y = .omp_data_i->y;
  .omp_data_o.5.x = .omp_data_i->x;
  .omp_data_o.5.a = &.omp_data_i->a;
  GOMP_task (main._omp_fn.1, &.omp_data_o.5, ...);

// main._omp_fn.1 (struct .omp_data_s.4 * .omp_data_i)
  y = .omp_data_i->y;
  x = .omp_data_i->x;
  a_p = .omp_data_i->a;
  *a_p = x + y;
```

Example: PRE (continued)

```
x = 2;
y = 3;

if (__property_parallel () && __property_firstprivate (x)
     && __property_firstprivate (y) && __property_shared (&a)
{
  if(__property_single ())
  {
    for (i = 0; i < N; ++i)
    if (__property_task () && __property_firstprivate (x)
         && __property_firstprivate (y) && __property_shared (&a))
    {
      a = x + y;
    }
  }
}

// use a
```

↓ This does not allow to optimize all the way ... ↓

Example: PRE (continued)

↓ This does not allow to optimize all the way ... ↓

```
if (__property_parallel () && __property_firstprivate (2)
     && __property_firstprivate (3) && __property_shared (&a)
{
  if(__property_single ())
  {
    for (i = 0; i < N; ++i)
    if (__property_task () && __property_firstprivate (2)
         && __property_firstprivate (3) && __property_shared (&a))
    {
      a = 5;
    }
  }
}

// use a
```

↓ Which yields ↓

```
#pragma omp parallel shared (a)
  #pragma omp single
    for (i = 0; i < N; ++i)
      #pragma omp task shared (a)
        a = 5;

  // use a
```

Conclusion

Implementation (on-going work)

- Work-streaming algorithm
  - task-level optimizations
  - improved cache locality
  - enables vectorization
- Runtime support for streaming
  - low overhead synchronization
  - low-cost deadlock detection scheme
  - debugging support
- Late expansion of OpenMP constructs
  - avoid obfuscation of code from early expansion
  - provide high-level user information to the optimization passes
  - enable optimizations based on static analysis

Builtin representation

- Specification is still incomplete for the behaviour of properties
- Abstraction of properties (eg. firstprivate $\rightarrow$ use)
- Free-lunch: serial optimizations become possible

Example: FMradio

```
// Implement PRE operations (delays).
#pragma omp task output (fm_qd_buffer << fm_qd_buffer_pre[maxtaps_minus_one]) private (i)
  for (i = 0; i < maxtaps_minus_one; ++i)
    fm_qd_buffer_pre[i] = 0;

#pragma omp task output (ffd_buffer << view_3[lp_3_taps_minus_eight]) private (i)
  for (i = 0; i < lp_3_taps_minus_eight; ++i)
    view_3[i] = 0;

  while ((16 == fread (read_buffer, sizeof(float), 16, input_file))) {

#pragma omp task firstprivate (read_buffer) output (fm_qd_buffer << view8[8])
    for (i = 0; i < 8; i++)
      fm_quad_demod (&fm_qd_conf, read_buffer[2*i], read_buffer[2*i + 1], &view8[i]);

      for (i = 0; i < 8; i++) {
#pragma omp task input (fm_qd_buffer >> view_11[1]) output (band_11)
        ntaps_filter_ffd (&lp_11_conf, 1, &view_11[diff_11], &band_11);

#pragma omp task input (fm_qd_buffer >> view_12[1]) output (band_12)
        ntaps_filter_ffd (&lp_12_conf, 1, &view_12[diff_12], &band_12);

#pragma omp task input (fm_qd_buffer >> view_21[1]) output (band_21)
        ntaps_filter_ffd (&lp_21_conf, 1, &view_21[diff_21], &band_21);

#pragma omp task input (fm_qd_buffer >> view_22[1]) output (band_22)
        ntaps_filter_ffd (&lp_22_conf, 1, &view_22[diff_22], &band_22);

#pragma omp task input (band_11, band_12, band_21, band_22) output (ffd_buffer)
        {
          subctract (band_11, band_12, &resume_1);
          subctract (band_21, band_22, &resume_2);
          multiply_square (resume_1, resume_2, &ffd_buffer);
        }
      }

#pragma omp task input (fm_qd_buffer >> view_2[8], ffd_buffer >> view_3[8]) output (band_2, band_3)
      {
        ntaps_filter_ffd (&lp_2_conf, 8, &view_2[diff_2], &band_2);
        ntaps_filter_ffd (&lp_3_conf, 8, view_3, &band_3);
      }
#pragma omp task input (band_2, band_3)
      {
        stereo_sum (band_2, band_3, &output1, &output2);
        output_short[0] = dac_cast_trunc_and_normalize_to_short (output1);
        output_short[1] = dac_cast_trunc_and_normalize_to_short (output2);
        fwrite (output_short, sizeof(short), 2, output_file);
      }
  }
```

Performance Evaluation

FMradio

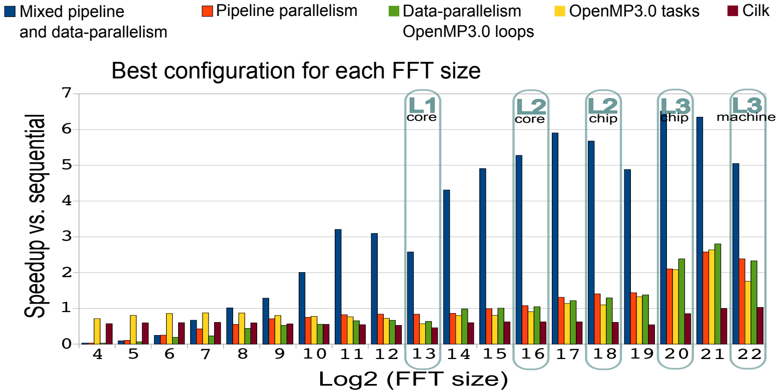
- high amount of data-parallelism, fairly well-balanced
- little effort to annotate with the extended OpenMP directives
- $\mathbf{12.6}\times$ speedup on 16 cores Opteron ($\mathbf{10.5}\times$ automatic code generation – $\mathbf{20\%}$)
- $\mathbf{18.8}\times$ speedup on 24 cores Xeon

IEEE802.11a

- complicated to parallelize, more unbalanced
- complex code refactoring is necessary to expose data parallelism
- annotating the program is straightforward to exploit pipeline parallelism
- annotating while enabling data-parallelism is difficult
- $\mathbf{13}\times$ speedup on 16 cores Opteron ($\mathbf{6}\times$ automatic code generation – $\mathbf{55\%}$)
- $\mathbf{14.9}\times$ speedup on 24 cores Xeon

Single FFT Performance

- **4.85**× speedup on 4 socket, 24 cores Xeon
- **6.5**× speedup on 4 socket, 16 cores Opteron



4-socket Opteron – 16 cores