

Improving GCC Infrastructure for Streamization

Antoni Pop, Sebastian Pop,
Harsha Jagasia, Jan Sjödin, Paul H. J. Kelly

Centre de Recherche en Informatique, Ecole des mines de Paris, France

Solutions Enablement Engineering, Advanced Micro Devices, Austin, Texas

Imperial College London, UK

June 18, 2008

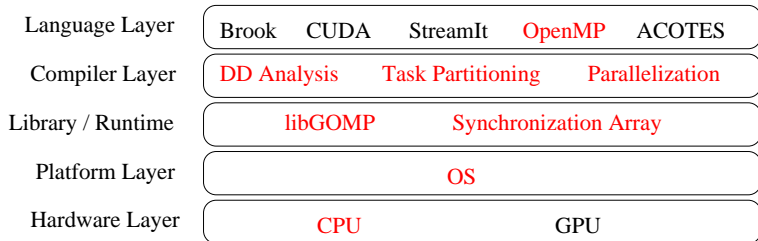
Outline

- ▶ what is a stream
- ▶ runtime stream support
- ▶ automatic streamization

contributions

- ▶ support for streams optimized for AMD Phenom™ 9550
- ▶ detecting producer/consumer patterns in the compiler

System Architecture for Stream Support



Frames vs. Streams: What is a frame?

a frame contains data that can be processed in any order:

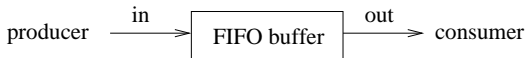
- ▶ Brook “streams”
- ▶ Ben Gaster’s “stream” proposal (IWOMP’08)

frames can be used as containers for parallel algorithms (no dependences between computations), but they could be unsafe for computations where the order and associativity is important:

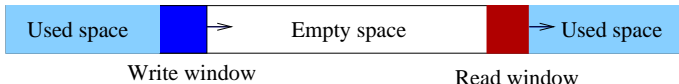
- ▶ floating point precision
- ▶ saturating computations
- ▶ non defined wrap around for signed integer in C/C++

Frames vs. Streams: What is a Stream?

streams encode the sequential behavior of computation, the data in a stream is consumed in the same order as produced



streams can be implemented as a concurrent lock-free queue (Lamport'83): circular buffer with read and write pointers



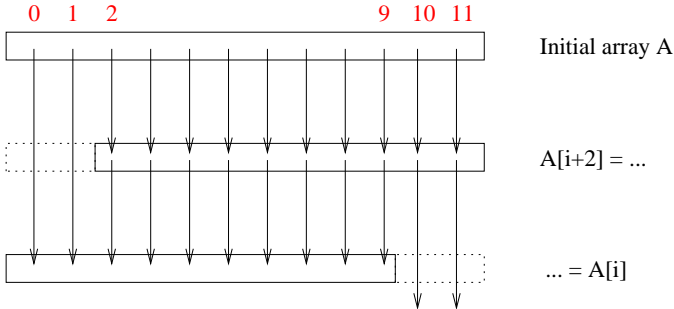
Detection of Stream Patterns

```
for(i = 0; i < N; i++) {
  A[i+2] = ...;
  ... = ... A[i] ...;
}
```

Partition →

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

split computations along flow dependences in producer/consumer



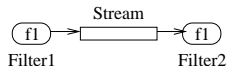
Flow Dependences to Stream

replace shared memory communication with stream communication

- ▶ avoids complex shared memory emulation
- ▶ explicit task comm for NUMA and heterogeneous systems

```
for(i = 0; i < N; i++) {  
    A[i+2] = ...;  
}
```

```
for(i = 0; i < N; i++) {  
    ... = ... A[i] ...;  
}
```

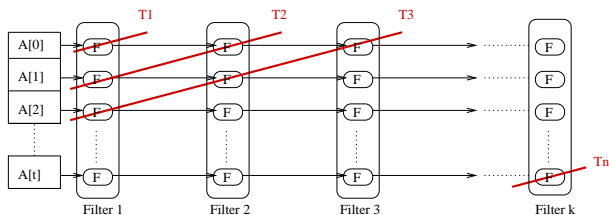


Stream →

```
s = gomp_stream_create(...);  
gomp_stream_push(s, A[0]);  
gomp_stream_push(s, A[1]);  
for(i = 0; i < N; i++) {  
    A[i+2] = ...;  
    gomp_stream_push(s, A[i+2]);  
}
```

```
for(i = 0; i < N; i++) {  
    ... = ... gomp_stream_head(s) ...;  
    gomp_stream_pop(s);  
}
```

Pipeline of Filters: Static Schedule (Hyperplanes of Lamport 1982)

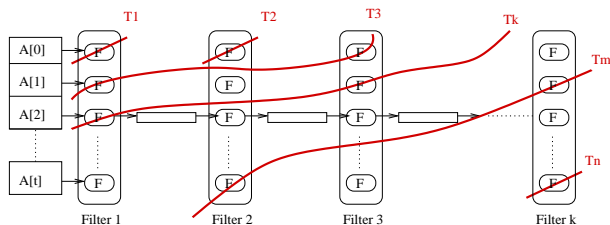


time

- ▶ T1: Filter1 processes $A[0]$
- ▶ T2: Filter2 processes $A[0]$, and in the same time Filter1 processes $A[1]$
- ▶ etc.

synchronization is needed for Filter2 to not process elements that Filter1 did not processed

Pipeline of Filters: Dynamic Schedule



- ▶ shared memory replaced by stream (FIFO) communication
- ▶ no synchronization needed: parallel schedule determined at runtime following the availability of elements in streams
- ▶ streams sequentialize computation and allow parallelism
- ▶ unconstrained schedule allow the OS to adapt at runtime for unbalanced workloads and different power processing elements

Evaluating Performance of Stream Support

outline for next slides

- ▶ sequential stream benchmarks
- ▶ streamized stream benchmarks
- ▶ variations on size of sliding windows
- ▶ variations on size of stream buffer

Sequential Stream Benchmarks

data from an array is processed then passed to a next task

```
int *A = (int *) malloc (32 * 1024 * 1024);
```

```
for(i = 1; i <= N; i++)  
  A[i] = WorkLoadk (A[i]);
```

```
for(i = 1; i <= N; i++)  
  A[i] = WorkLoadk (A[i]);
```

can vary the number and load of tasks

- ▶ *WorkLoad₁*: euclidean distance $\sqrt{x * x + y * y}$
- ▶ *WorkLoad_k*: *WorkLoad₁* iterated k times

Streamized Stream Benchmarks

- ▶ data processed: image of 32MB
- ▶ stream of integers of size $1000 * 64B$
- ▶ 64B read and write sliding windows


```
int *A = (int *) malloc (32 * 1024 * 1024);
gomp_stream s = gomp_stream_create (4, 1000, 64);
#pragma omp parallel sections num_threads (2)
{
```

```
#pragma omp section
  /* Producer task. */
```

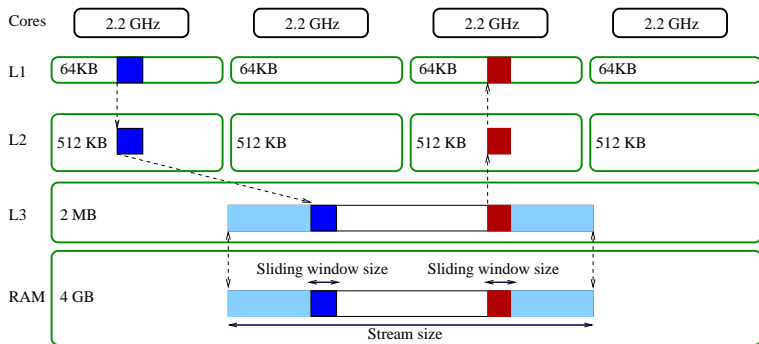
```
{
  int i;
  for(i = 1; i <= N; i++) {
    elt = WorkLoadk (A[i]);
    gomp_stream_push (s, elt);
  }
  gomp_stream_set_eos (s);
}
}
```

```
#pragma omp section
  /* Consumer task. */
```

```
{
  int i;
  for(i = 1; i <= N; i++) {
    elt = gomp_stream_head (s);
    A[i] = WorkLoadk (elt);
    gomp_stream_pop (s);
  }
  gomp_stream_destroy (s);
}
```



Architectural Setup: AMD Phenom™ 9550



- ▶ L1 cache line size: 64 B
- ▶ size of sliding windows \geq L1 cache line: avoids cache trashing
- ▶ large stream buffers
 - ▶ allow more work without waits for free space
 - ▶ may flush data out of the caches

Variations on Workload and Number of Parallel Workers

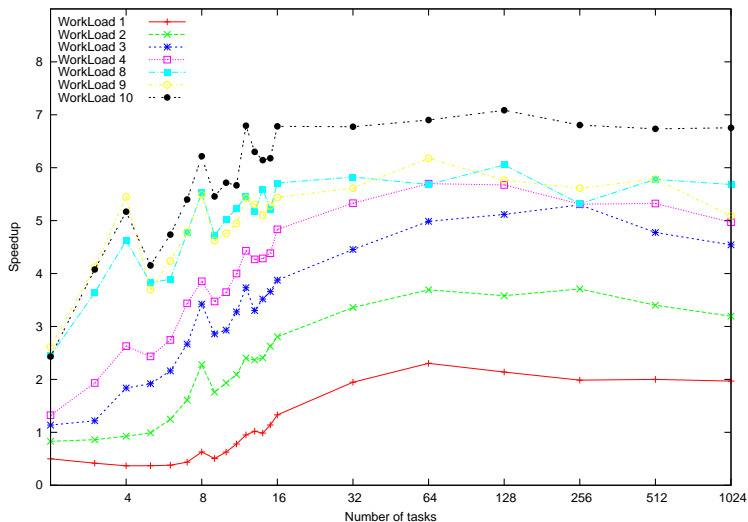
experimental setup

- ▶ stream size: 1000 * 64B
- ▶ sliding window size: 64B
- ▶ length of tasks pipeline: varies from 2 to 1024 tasks
- ▶ workload per task: varies from 1 to 10

main results

- ▶ speedup of $7 = \frac{t_{sequential}}{t_{stream}}$ on AMD Phenom quad-core: cache acceleration, loop fusion at runtime and parallelization
- ▶ system overhead hidden by parallelism and communication accelerated by L3 cache: streams stay in the L3 cache
- ▶ higher speedups for high workloads per task or long pipelines
- ▶ scales well for huge pipelines (1024 tasks)

Variations on Workload and Number of Parallel Workers



Variations on Stream Size and Sliding Window Size

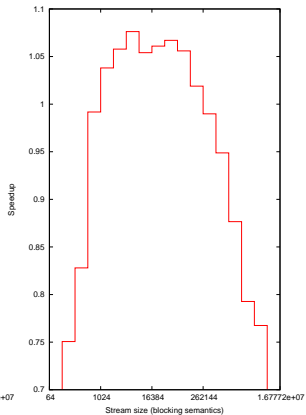
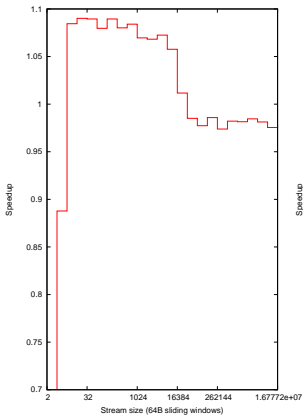
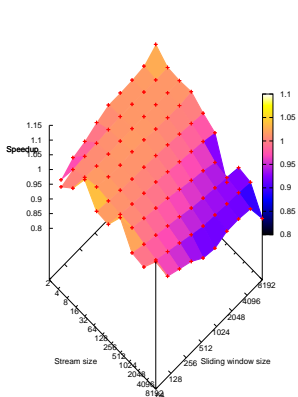
experimental setup

- ▶ stream size: varies from $2 * 64B$ to $16M * 64B$
- ▶ sliding window size: varies from $64B$ to $16MB$
- ▶ length of tasks pipeline: 2
- ▶ workload per task: 3

main results

- ▶ best speedup: 1.09 for 16 to 1024 sliding windows of $64B$
- ▶ streams bigger than L3 cache size not profitable
- ▶ streams shorter than $16 * 64B$ not profitable: too much synchronization

Variations on Stream Size and Sliding Window Size



Conclusion and Future Work

results

- ▶ streams with sequence semantics can be useful for parallelization of sequences of tasks
- ▶ enough workload is needed to hide overhead of OS
- ▶ prototype for task split and streamization in graphite branch

future work

- ▶ auto-streamize straight line code in GCC
- ▶ integrate streamization in the code generator of graphite
- ▶ improve runtime stream support to hide OS overheads

Questions