A Proposal for lastprivate Clause on OpenMP task Pragma

Antoniu Pop and Sebastian Pop

Centre de Recherche en Informatique, MINES ParisTech, France
apop@cri.ensmp.fr

Compiler Performance Engineering, Advanced Micro Devices, Austin, Texas
sebastian.pop@amd.com

Abstract. Several implementations extend OpenMP with pragmas for programming heterogeneous systems using stream primitives. This paper surveys some of these extensions and then provides a minimal extension to the OpenMP3.0 standard to support data streams between tasks. We present a prototype implementation of the proposed extension in the GCC compiler and its runtime GOMP library.

1 Introduction

For programming devices with non uniform memory spaces, programmers use data transmission primitives, such as MPI's send and recv. As we will see in this paper, OpenMP3.0 could support the notion of message passing via the firstprivate and lastprivate constructs. The missing part for the task pragma defined in OpenMP3.0 is the specification of the lastprivate clause that would allow the definition of outputs from a task using private memory. This extension of OpenMP does not change the shared memory model of OpenMP, but intends to permit the translation of tasks and their communications on distributed memory systems.

The paper starts by presenting previous work on manual techniques for writing code with data streams. Section 3 presents our extension to OpenMP3.0, and Section 4 describes some of the optimizations that this extension would enable. Finally, Section 5 describes and evaluates the implementation of our prototype based on GCC4.4 compilers.

2 Related Work

Stream programming has recently attracted a lot of attention as an alternative to other forms of parallel programming that offers an improved programmability and may, to a certain extent, reduce the severity of the memory wall. Many languages and libraries are available for programming stream applications. Some are general purpose programming languages that hide the underlying architecture's specificities, while others are primarily graphics processing languages, or shading languages. Some hardware vendors also propose low-level interfaces for their GPUs.

The StreamIt language [4] is an explicitly parallel programming language that implements the Synchronous Data Flow (SDF) programming model. It contains syntactic constructs for defining programs structured as task graphs. Tasks contain Java-like code that is executed in a sequential mode. StreamIt provides three interconnection modes: the Pipeline allows the connection of several tasks in a straight line, the SplitJoin allows for nesting data parallelism by dividing the output of a task in multiple streams, then merging the results in a single output stream, and the FeedbackLoop allows the creation of streams from consumers back to producers. The channels connecting tasks are implemented either as circular buffers, or as message passing for small amounts of control information.

The Brook language [5] provides language extensions to C with single program multiple data (SPMD) operations that work on streams, i.e. control flow is synchronized at communication/synchronization operations. Streams are defined as collections of data that can be processed in parallel. For example: "`float s<100>;`" is a stream of 100 independent floats. User defined functions that operate on streams are called kernels and use the "kernel" keyword in the function definition. The user defines input and output streams for the kernels that can execute in parallel by reading and writing to separate locations in the stream. Brook kernels are blocking: the execution of a kernel must complete before the next kernel can execute. This is the same execution model that is available on graphics processing units (GPUs): a task queue contains the sequence of shader programs to be applied on the texture buffers. The *CUDA* infrastructure from NVIDIA [6] is similar to Brook, but also invites the programmer to manage local scratchpad memory explicitly: in CUDA, a block of threads, assigned to run in parallel on the same core, share access to a common scratchpad memory. CUDA is lower level than Brook from a memory control point of view. The key difference is that CUDA has explicit management of the per-core shared memory. Brook was designed for shaders: it produces one output element per thread, any element grouping is done using input blocks reading from main memory repeatedly.

The ACOTES project [1] proposes extensions to the OpenMP3.0 standard that can be used for manually defining complete task graphs, including asynchronous communication channels: it adds new constructs and clauses as for example a new task pragma with clauses for defining inputs and outputs [2, 3, 8]. The implementation of the ACOTES extensions to OpenMP3.0 includes two parts: the compiler part translates the pragma clauses to calls to a runtime library extending the OpenMP library. The ACOTES extensions, are an attempt to make communication between tasks explicit. Channels can be implemented on top of shared memory as well as on top of message passing. ACOTES extensions can be classified MIMD, as several tasks can execute in parallel on different data streams. This aims to shift the memory model of OpenMP from shared memory to distributed memory for the task pragmas. The resulting ACOTES programming model can be compared to the Brook language: these languages both provide the notion of streams of data flowing through processing tasks that can potentially contain control flow operations. The main difference between these two programming languages is in their semantics. In the execution model of a Brook task, the task is supposed to process all the data contained in the stream before executing another task. The tasks in the ACOTES semantics are non-blocking: the execution of a task can proceed as soon as some data is available in its input streams. The main limitation of the Brook language is due to the intentionally blocking semantics that follows the constraints of the target hardware, i.e. GPUs, where the executing tasks have to be loaded on the GPU, an operation that has a non-negligible cost. The design of the Brook language and of CUDA follow these constraints, restricting the expressiveness of the language, intentionally. The ACOTES programming model does not contain these limitations and the runtime library support of the ACOTES streams can dynamically select the blocking semantics of streams to fit the cost constraints of the target hardware.

Another interesting approach to generate the data transmission towards the accelerator boards is that of the CAPS enterprise: codelets are functions [11] whose parameters can be marked with input, output or inout. The codelets are intended to be executed remotely after the input data has been transmitted.

Some recent developments of the ICC compiler implement OpenMP extensions for generating code for processors with accelerators: data can be sent, retrieved and potentially left in the memory of the remote accelerator.

In the next section, we describe a minimal change to the OpenMP3.0 task pragmas to allow the specification of data channels between tasks.

3 OpenMP extension for enabling streaming

We propose an extension of the OpenMP3.0 standard to allow for lastprivate clauses to be used on task constructs. This extension does not change the semantic of the lastprivate clause. As defined by the OpenMP3.0 standard, the lastprivate clause provides a superset of the functionality provided by the private clause.

3.1 Description of the extension

We extend the OpenMP semantic for task constructs: a list item that appears in a lastprivate(*list*) clause of a task construct will be assigned the last value within the task upon completion. The list of modifications required to the standard is given in Appendix A.

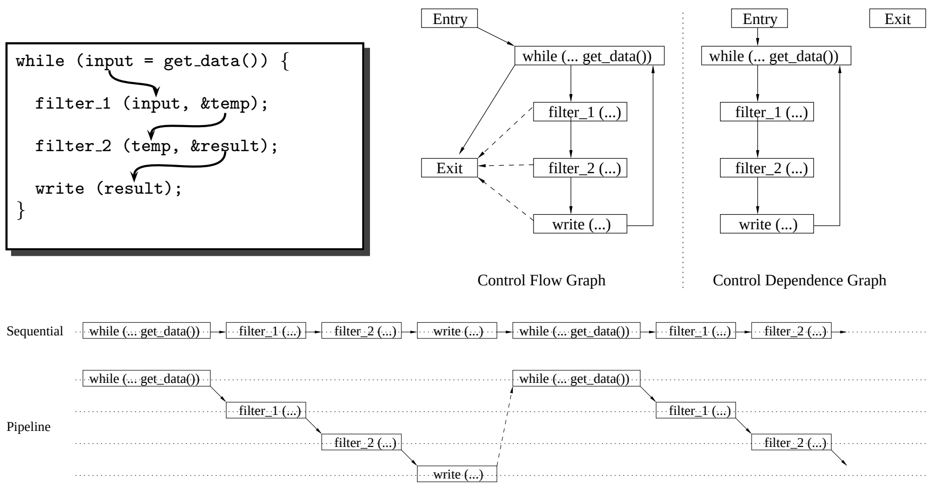
Due to the semantic of the task construct, an implicit synchronization point is created before assigning the original list item upon completion of the task to avoid data races. This behaviour can be altered by optimizations as we will see in Section 4.

The consequence of this synchronization is that a thread that encounters a task construct with a lastprivate clause will suspend the current task region, which may not be resumed until the generated task is completed. This ensures that the value of the list items that appear in lastprivate(*list*) clauses is properly updated before the encountering thread proceeds. The synchronization will therefore prevent concurrency and void any benefit of generating a task. In the case where no optimization is possible, such a task construct should ultimately be ignored and the code executed sequentially.

3.2 Motivation for the extension

Our primary motivation for the introduction of this extension is to allow tasks to explicitly assume the roles of producers and consumers of data. This is often the case implicitly when tasks use shared clauses and thus communicate through shared memory, but the synchronization requirements make this impractical.

The semantic of firstprivate and lastprivate clauses is very close to what the Stream Programming Model of the ACOTES project [3] calls input and output clauses. The firstprivate clause corresponds to data that is consumed by the task (flows in), while the lastprivate clause corresponds to data that is produced by the task (flows out). The knowledge of data flow between tasks helps the static analysis.



```
while (input = get_data()) {

  filter_1 (input, &temp);

  filter_2 (temp, &result);

  write (result);
}
```

Fig. 1. A simple pipeline of filters (left), where arrows represent flow dependences, and the corresponding *Control Flow Graph (CFG)*, *Control Dependence Graph (CDG)* and the execution trace.

In order to understand the necessity of the extension and the related optimizations, let us consider the example on Figure 1. This example is a simple while loop where some input data is read and fed to a pipeline of filters, with a final write of the result. Though it may appear that this pipeline is easy to detect with static analysis, without even needing OpenMP annotations, the control dependences will inhibit any such automatic optimization as long as it is impossible to determine whether the filter functions can branch out of the loop (dashed edges of the CFG). As we can see on the CDG, there is no room for concurrency and the resulting execution trace would be serialized by the dashed edges representing control dependences.

However, the semantic of the OpenMP3.0 task constructs is such that the responsibility for ensuring that the filters are well-behaved is left to the programmer (for example that they do not throw exceptions or catch them appropriately). The code on Figure 2 is an example of how this pipeline of filters should be annotated. The CFG shows that, because of the restrictions on the code that can be inside a task construct, there are no longer edges from the filters to the exit node, which in turn translates on the CDG in significantly fewer and less restrictive control dependences. The possible execution trace shows that concurrency is no longer impossible.

Though the execution trace shows that pipeline parallelism is now possible, we cannot remove the lastprivate clauses because the data dependences would no longer be enforced. The only alternative would be to replace them with shared clauses and manual synchronization.

We will now propose a very simple expansion scheme for lastprivate clauses that provides for an easy and correct implementation.

3.3 Expansion of lastprivate clauses for task constructs

Before considering any optimizations, let us propose a simple expansion rule for lastprivate clauses on OpenMP task constructs. We will follow the earlier proposed semantic and introduce a synchronization point before assigning the new value to any item appearing on the lastprivate clause.

This expansion scheme is only meant as a way of trivially ensuring the validity of the generated code in non-optimizing compilers. It is by no means the objective of this extension as it results in no performance improvement and in most cases would only degrade performance. The expansion scheme has already been implemented in GCC and is freely available on the streamOMP branch.

The implementation is quite straightforward and only marginally alters the original code. In the GCC implementation of OpenMP, the expansion of firstprivate and lastprivate clauses is implemented by copying the variables in dedicated data structures for copy-in (_omp_data_i)and copy-out (_omp_data_o) that are accessible to the task for reading input data and storing output data. We do not modify this, but just add the semaphore synchronization.

When a lastprivate*(list)* clause appears on a task construct, for all items $x \in list$ copy-out code needs to be generated. In the outlined code of the task, the copy statements $\{\forall\ x \in list,\ omp\_data\_o.x = x;\}$ will be added at the end of the task as well as a synchronization post call. In the original context, after the task generation call, a corresponding synchronization wait call as well as the statements $\{\forall\ x \in list,\ x = omp\_data\_o.x;\}$ are inserted.

```
#pragma omp parallel
  {
#pragma omp single
    {
      while (input = get_data())
        {
#pragma omp task firstprivate (input) lastprivate (temp)
          filter_1 (input, &temp);

#pragma omp task firstprivate (temp) lastprivate (result)
          filter_2 (temp, &result);

#pragma omp task firstprivate (result)
          write (result);

        }
    }
  }
```

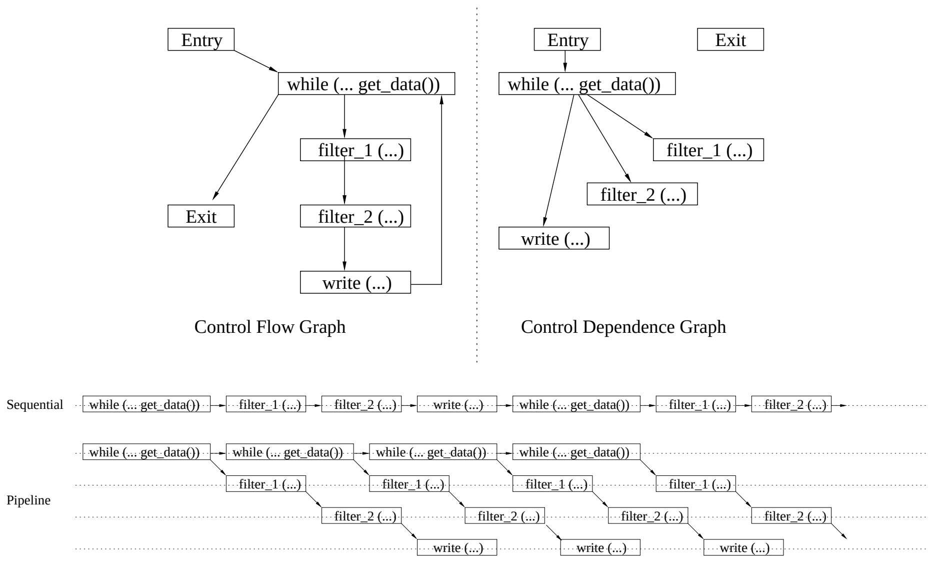
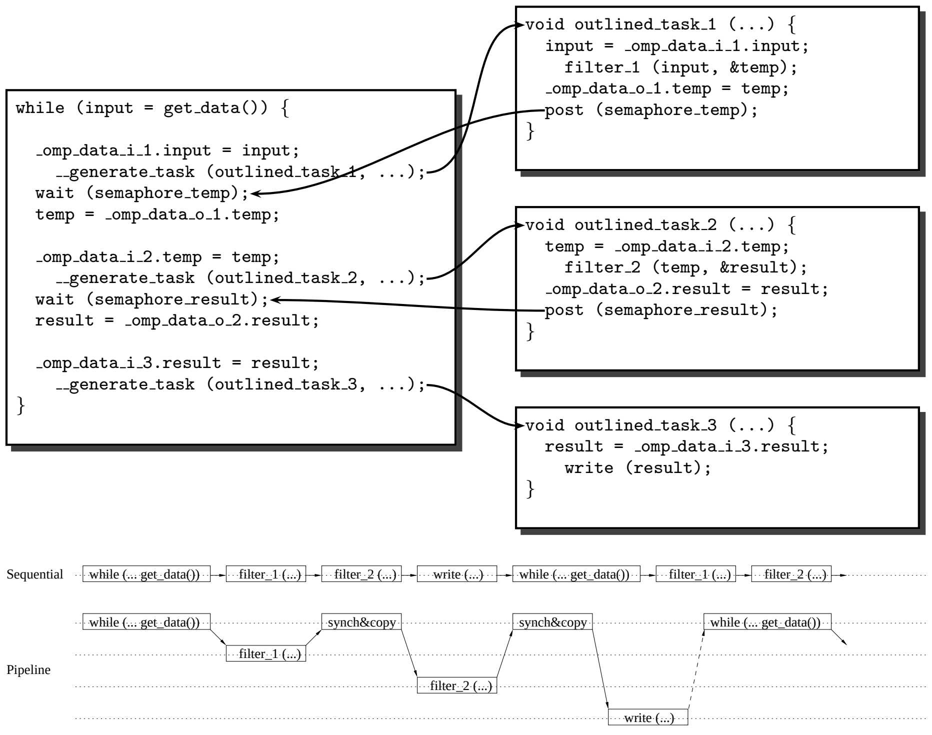


Fig. 2. Pipeline of filters with OpenMP annotations and the corresponding *Control Flow Graph (CFG)*, *Control Dependence Graph (CDG)* and a possible execution trace. The flow dependences marked by arrows in the code must be satisfied.

In order to illustrate the diverse transformations we will use the same example of filter pipeline. We first apply the expansion to the code of the example on Figure 1. The resulting code is presented on Figure 3. The strong synchronization introduced all but serializes the execution. Despite the fact that the outlined code of the two tasks run on different threads, the trace remains similar to the sequential trace with additional overhead. All existing dependences are



```
while (input = get_data()) {

  _omp_data_i_1.input = input;
    __generate_task (outlined_task_1, ...);
  wait (semaphore_temp);
  temp = _omp_data_o_1.temp;

  _omp_data_i_2.temp = temp;
    __generate_task (outlined_task_2, ...);
  wait (semaphore_result);
  result = _omp_data_o_2.result;

  _omp_data_i_3.result = result;
    __generate_task (outlined_task_3, ...);
}
```

```
void outlined_task_1 (...) {
  input = _omp_data_i_1.input;
    filter_1 (input, &temp);
  _omp_data_o_1.temp = temp;
  post (semaphore_temp);
}
```

```
void outlined_task_2 (...) {
  temp = _omp_data_i_2.temp;
    filter_2 (temp, &result);
  _omp_data_o_2.result = result;
  post (semaphore_result);
}
```

```
void outlined_task_3 (...) {
  result = _omp_data_i_3.result;
    write (result);
}
```

Fig. 3. Code generated by the expansion of the task directives with firstprivate and lastprivate clauses. The execution trace shows that there is nothing to be gained without further work.

preserved, which ensures the validity of the transformation. Of course in this case, no concurrency is possible.

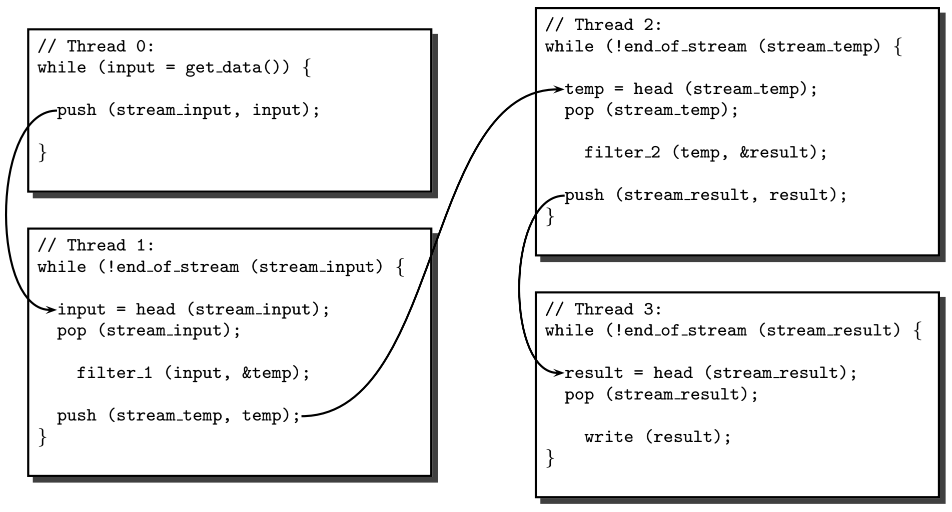
However, in the next section, we will see that in some cases we can do better than to simply serialize the execution of tasks with lastprivate clauses.

4 Compiler optimizations

This section describes some of the optimizations that the extension we propose would enable. These optimizations are currently being implemented in the GCC compiler. Though no preliminary results are available for the automatic generation of optimized code at this time, the evaluation Section 5 presents some results obtained by hand-optimizing the code, thus demonstrating the potential of the automatic optimization being implemented.

The most important optimization enabled by this extension is to allow tasks to communicate through streams. We will refer to this optimization as streamization. As we have seen, the use of firstprivate and lastprivate clauses makes the data flow between tasks explicit. It is possible, with some static dependence analysis, to build a taskgraph and identify the channels of communication between tasks.

We will again use the example on Figures 1 and 2 to illustrate the optimizations. First of all, we note that no data-parallelism can readily be exploited because of the way input data is accessed and task parallelism would be very hard, or expensive, to synchronize. Task parallelism would be a possibility, but synchronization is required to satisfy the flow dependences. The only appropriate form of parallelism in this case is pipelining. Our objective is to generate the code presented on Figure 4. In this example, we use a stream communication library where streams behave as blocking FIFO queues, therefore ensuring synchronization of flow dependences. Details on the semantic and the implementation of the streaming library can be found in [10]. The implementation is available in the GOMP library of GCC. An alternative to stream communication would be to use MPI sends and recvs for distributed platforms, though this requires that tasks access no shared data. The very interesting work by Millot et al. [9] explores the idea of compiling OpenMP annotated code for distributed systems.



```
// Thread 0:
while (input = get_data()) {

  push (stream_input, input);

}
```

```
// Thread 1:
while (!end_of_stream (stream_input) {

  input = head (stream_input);
  pop (stream_input);

    filter_1 (input, &temp);

  push (stream_temp, temp);
}
```

```
// Thread 2:
while (!end_of_stream (stream_temp) {

  temp = head (stream_temp);
  pop (stream_temp);

    filter_2 (temp, &result);

  push (stream_result, result);
}
```

```
// Thread 3:
while (!end_of_stream (stream_result) {

  result = head (stream_result);
  pop (stream_result);

    write (result);
}
```

Fig. 4. Resulting code after streamization. Generating this code is the objective of this optimization, though the current implementation in GCC has not yet reached this point. The execution trace is similar to that on Figure 2.

The stream communication is useful here because it buffers (privatizes) the values produced until the consumer can use them. This means that the flow dependences they enforce are somewhat relaxed. The proposed OpenMP annotated code uses our extension as a means of synchronizing the tasks. This is necessary because of the flow dependences present between the filters (e.g., filter_1 writes to temp and filter_2 reads that value). The presence of firstprivate and lastprivate clauses makes explicit the need to enforce these dependences. If we were to consider writing an equivalent code without resorting to the usage of lastprivate clauses on task constructs, we would need to use shared clauses and the appropriate manual synchronization. If the programmer does not annotate the code with OpenMP directives, then static analysis may fail due to the lack of knowledge on the side-effects of the filter functions.

In order to generate the code on Figure 4 from the OpenMP annotated code on Figure 2, we need some static analysis, both for control and data dependences, to ensure that the transformation is valid. If for example we modify the code of the example on Figure 2 and add some code between the two task constructs containing the filters and that code accesses the variable temp, then the streamization is no longer possible.

5 Evaluation of the GCC Prototype

The current implementation of this extension in GCC is still limited to the basic expansion of lastprivate clauses and generates the serializing synchronization described in Section 3.3. As the implementation of the optimizations is still under way, we will first exmine the slowdowns incurred compared to sequential execution in the case where no optimizations are performed. We will also show, as a motivation for the optimizations under development, what can be gained from streamizing the code. In this second experiment, we write the code as it would be generated by an optimizing compiler, with no additional manual optimizations.

As a basis for evaluation, we will consider a kernel extracted from the GNU Radio project [7]. This kernel was originally extracted by Marco Cornero, from STMicroelectronics, and further adapted for streaming by David Rodenas-Pico, from the Barcelona Supercomputing Center, for the needs of the ACOTES project [1]. We also slightly modified the kernel for our experiments, by annotating it with OpenMP task pragmas with firstprivate and lastprivate clauses. The main loop of the annotated kernel is presented on Figure 6.

The evaluation of all the benchmarks presented in this paper is performed using a modified version of GCC4.4 available in the streamOMP branch. The experimental setup consists of the two platforms presented, along with the experimental results, on Figure 5. The implementation of the streaming library takes advantage of the memory hierarchy by aggregating communication in reading/writing windows. These windows are at least of the size of an L1 cache line which reduces false sharing and improves performance [10].

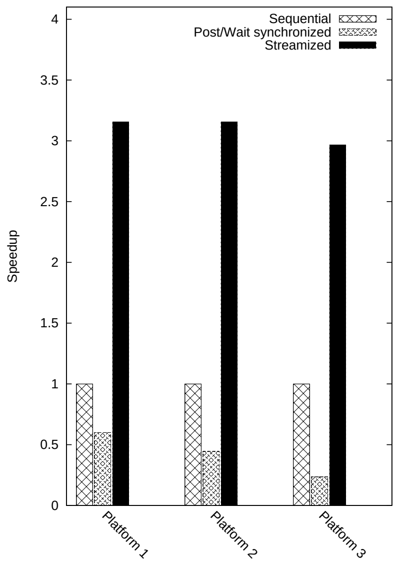
As we expected, the slowdowns incurred in the case of the basic expansion scheme, where semaphore post/wait synchronization is used, are quite high. The execution is between 1.6 and 4.2 times slower than the sequential counterpart. Though it may be possible to reduce the overhead responsible for these slowdowns, we consider that, when it is impossible to optimize, task constructs with lastprivate clauses should be inlined, or simply not expanded. As the thread

Platform 1: Dual AMD Opteron^{TM} Barcelona B3 CPU 8354 with 4 cores at 2.2GHz, running under Linux kernel 2.6.18, and the following characteristics of the memory hierarchy:

- L1 cache line size: 64 B
- 64 KB per core L1 cache
- 512 KB per core L2 cache
- 2 MB per chip shared L3 cache
- 16 GB RAM

Platform 2: IBM JS22 Power6 with 4 cores, each two-way SMT, at 4GHz, running under Linux kernel 2.6.16. Memory characteristics:

- L1 cache line size: 128 B
- 64 KB L1 cache
- 2 MB per core L2 cache
- 8 GB RAM



Platform 3: Intel® Core^{TM}2 Quad CPU Q9550 with 4 cores at 2.83GHz, running under Linux kernel 2.6.27, and the following characteristics of the memory hierarchy:

- L1 cache line size: 64 B
- 32 KB per core L1 cache
- 2 independent 6 MB shared L2 caches
- 4 GB RAM

Fig. 5. Speedups to sequential execution obtained on the GNU Radio kernel presented on Figure 6. We evaluate the speedups of the automatically generated post/wait synchronization and the hand-streamized code on both test platforms.

encountering the task waits for its completion, there is nothing to gain from executing it on another thread and synchronizing upon completion.

On the other hand, the streamized code shows reasonably high speedups. On average, the execution of the hand-streamized code is more than three times faster than the sequential version on all platforms. Such results are a strong incentive to continue the development of the streamization pass in GCC. We note that the load balance is not perfect yet as only two of the thirteen filters present in the application have a high arithmetic intensity. This results in equivalent speedups on all platforms despite the fact that platforms 1 and 2 have 8 hardware threads whereas platform 3 only has 4 hardware threads.

We believe that extending the OpenMP standard by allowing lastprivate clauses for task constructs is a minimal and efficient way of exploiting pipeline parallelism in OpenMP.

It also provides for some interesting optimization opportunities. The streaming library can be easily implemented over a message passing interface, therefore allowing tasks that do not communicate through shared memory (i.e., no shared clause) to be executed on processors that do not have access to the main thread's memory.

6 Conclusion

We proposed to extend the OpenMP3.0 standard by defining the semantics of the lastprivate clause to the task pragama. This is a natural way to extend the OpenMP3.0 specification that already defines the semantics of the lastprivate clause in the context of other pragmas. Furthermore, we focused on minimizing the changes needed to the current standard, to minimize the risk of errors introduced in the standard and the ease of implementation in compilers.

7 Acknowledgments

We would like to thank our colleagues from the ACOTES project for the fruitful discussions on this topic.

References

1. ACOTES: Advanced Compiler Technologies for Embedded Streaming. `http://www.hitech-projects.com/euprojects/ACOTES/`.
2. ACOTES deliverable d2.1. `http://www.hitech-projects.com/euprojects/ACOTES/deliverables/deliverable_D2.1_v1.0.pdf`.
3. ACOTES deliverable d3.2. `http://www.hitech-projects.com/euprojects/ACOTES/deliverables/acotes-d3.2-final.pdf`.
4. The StreamIt language. `http://www.cag.lcs.mit.edu/streamit/`.
5. The Brook Language. `http://graphics.stanford.edu/projects/brookgpu/lang.html`.
6. The CUDA Language. `http://www.nvidia.com/object/cuda_home.html`.
7. The GNU Radio project. `http://www.gnu.org/software/gnuradio/`.
8. P. Carpenter, D. Rdenas, X. Martorell, A. Ramrez, and E. Ayguad. A streaming machine description and programming model. In S. Vassiliadis, M. Berekovic, and T. D. Hmlinen, editors, *SAMOS*, volume 4599 of *Lecture Notes in Computer Science*, pages 107–116. Springer, 2007.
9. D. Millot, A. Muller, C. Parrot, and F. Silber-Chaussumier. Step: A distributed openmp for coarse-grain parallelism tool. In R. Eigenmann and B. R. de Supinski, editors, *IWOMP*, volume 5004 of *Lecture Notes in Computer Science*, pages 83–99. Springer, 2008.
10. A. Pop, S. Pop, H. Jagasia, J. Sjödin, and P. H. J. Kelly. Improving GNU compiler collection infrastructure for streamization. In *Proceedings of the 2008 GCC Developers' Summit*, pages 77–86, 2008. `http://www.gccsummit.org/2008`.
11. S. B. R. Dolbeau and F. Bodin. Hmpp: A hybrid multi-core parallel programming environment. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, 2007.

```
#pragma omp parallel
  {
#pragma omp single
    {
      while (16 == fread (read_buffer, sizeof (float), 16, input_file))
      {
        for (i = 0; i < 8; i++)
          {
            pair.first = read_buffer[i*2];
            pair.second = read_buffer[i*2 + 1];

#pragma omp task firstprivate (pair, fm_qd_conf) lastprivate (fm_qd_value)
            fm_quad_demod (&fm_qd_conf, pair.first, pair.second, &fm_qd_value);

#pragma omp task firstprivate (fm_qd_value, lp_11_conf) lastprivate (band_11)
            ntaps_filter_ffd (&lp_11_conf, 1, &fm_qd_value, &band_11);

#pragma omp task firstprivate (fm_qd_value, lp_12_conf) lastprivate (band_12)
            ntaps_filter_ffd (&lp_12_conf, 1, &fm_qd_value, &band_12);

#pragma omp task firstprivate (band_11, band_12) lastprivate (resume_1)
            subctract (band_11, band_12, &resume_1);

#pragma omp task firstprivate (fm_qd_value, lp_21_conf) lastprivate (band_21)
            ntaps_filter_ffd (&lp_21_conf, 1, &fm_qd_value, &band_21);

#pragma omp task firstprivate (fm_qd_value, lp_22_conf) lastprivate (band_22)
            ntaps_filter_ffd (&lp_22_conf, 1, &fm_qd_value, &band_22);

#pragma omp task firstprivate (band_21, band_22) lastprivate (resume_2)
            subctract (band_21, band_22, &resume_2);

#pragma omp task firstprivate (resume_1, resume_2) lastprivate (ffd_value)
            multiply_square (resume_1, resume_2, &ffd_value);

            fm_qd_buffer[i] = fm_qd_value;
            ffd_buffer[i] = ffd_value;
          }

#pragma omp task firstprivate (fm_qd_buffer, lp_2_conf) lastprivate (band_2)
          ntaps_filter_ffd (&lp_2_conf, 8, fm_qd_buffer, &band_2);

#pragma omp task firstprivate (ffd_buffer, lp_3_conf) lastprivate (band_3)
          ntaps_filter_ffd (&lp_3_conf, 8, ffd_buffer, &band_3);

#pragma omp task firstprivate (band_2, band_3) lastprivate (output1, output2)
          stereo_sum (band_2, band_3, &output1, &output2);

#pragma omp task firstprivate (output1, output2, output_file, text_file)
          {
            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);
            fprintf (text_file, "%-10.5f %-10.5f\n", output1, output2);
          }
      }
    }
  }
```

Fig. 6. Kernel extracted and adapted from the GNU Radio project [7] with OpenMP annotations. The lastprivate clauses on tasks enable streamization.

A Proposed changes to the OpenMP3.0 standard

We propose the following changes to the OpenMP3.0 standard:

– In Section 2.7 task Construct, add the clause lastprivate(*list*).
– In Section 2.9.3.5 lastprivate clause, add the semantics of lastprivate used on a task pragma: a list item that appears in a lastprivate(*list*) clause of a task construct will be assigned the last value within the task upon completion.