Improving GNU Compiler Collection Infrastructure for Streamization

Antoniu Pop
Centre de Recherche en Informatique, Ecole des mines de Paris, France
apop@cri.ensmp.fr

Sebastian Pop, Harsha Jagasia, Jan Sjödin
Solutions Enablement Engineering, Advanced Micro Devices, Austin, Texas
firstname.lastname@amd.com

Paul H J Kelly
Imperial College of London, UK
p.kelly@imperial.ac.uk

Abstract

GNU Compiler Collection (GCC) needs a strategy to support future multicore architectures, which will probably include heterogeneous accelerator-like designs with explicit management of scratchpad memories. Some have further restrictions; for example, SIMD has limited synchronization capabilities. Some platforms will probably offer hardware support for streaming, transactions, and speculation.

The purpose of this paper is to survey and evaluate some automatic and manual techniques for improving support for such targets in GCC. We focus on translation of sequential code for such platforms, *i.e.*, the translation to task graphs and their communication and memory access operations. The paper provides an evaluation of the communication library support on an AMD Phenom^{TM} X4 9550 quad-core processor. We use these experiments to tune the automatic task partitioning algorithm implemented in GCC. The paper concludes with recommendations for strategic developments of GCC to support a stream programming language and improve the automatic generation of streamized tasks.

1 Introduction

Several of the popular programming languages are scalar sequential languages (for example C, C++, Fortran, Java). Generating SIMD and MIMD code from these languages is challenging, but there are techniques that we present in Sections 2 and 3 that can generate efficient parallel code. In all the techniques that we present, the transfer of data between tasks happens via a communication channel, called stream.

We describe manual and automatic techniques that allow the parallelization of communicating tasks. The manual techniques that we present include extensions to languages, such the Brook language [4] that extends the syntax of the C language, and hints, such as the OpenMP pragmas [6], that directly convey information from the programmer to the compiler. Manual techniques can be more effective than automatic techniques in describing parallelism and communications, but the automatic approach is better suited to large legacy codes, as it does not require modifications of the source code. Section 3 presents an automatic technique for detecting parallel tasks with communication that we implemented in GCC, together with a runtime library support for streams of data. We provide an evaluation of the stream implementation that shows the cost of executing multiple communicating tasks in parallel.

The paper starts by presenting previous work in the automatic and manual streamization. The next section presents GCC's infrastructure for supporting automatic streamization, and the last sections define improvements to this infrastructure: first the static analysis improvements, then improvements to multi-task code generation and the interaction between the compiler and runtime libraries.

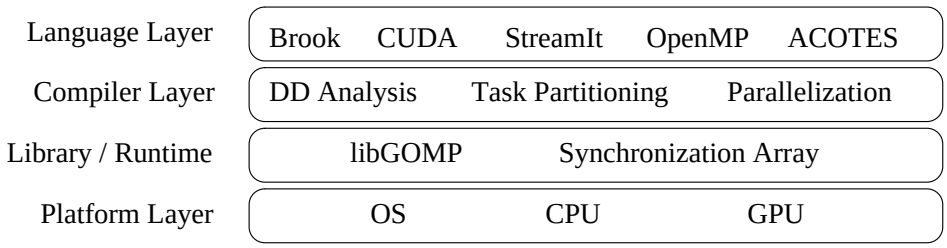


Figure 1: Techniques for streamization described in the related work section

2 Related Work

Figure 1 shows a software stack providing support for streaming computations: the language layer allows programmers to express the parallelism of the application, the techniques in the compiler layer automatically extract the parallelism, and the runtime library and platform layers provide execution support for streaming computations. The survey in this section is not intended to be an encyclopaedic review of related research, but instead a selective analysis of key points of reference.

2.1 Language Layer

The language layer provides a syntactic interface to the underlying layers.

- The `Brook` language [4] 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.

- `CUDA` [5] 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 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 `StreamIt` language [3] 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 Split allows a task to have more than one output or input streams; 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 low amounts of information.

- The `OpenMP` standard [6] extends the C, C++, and Fortran languages with pragmas for specifying parallel constructs, such as loops or sequential blocks of code. The compiler translates the `OpenMP` pragmas into calls to a threading runtime library assuming a shared memory model. `OpenMP` supports `SPMD` loop parallelism well, and task-parallelism partially. The model completely hides all the communication in the program, though programmers can mark some variables as private rather than shared. Similarly, reduction variables are handled in a declarative way that leaves the compiler to choose the implementation strategy. Version 3.0 of the `OpenMP` standard allows nested parallelism, and defines the notion of tasks [8]. Tasks in `OpenMP` enhance the support for task parallelism, and in particular pipelines; however, sharing and

communication are still hidden.

- The ACOTES project [1] proposes extensions to the OpenMP 3.0 standard that can be used for manually defining complete task graphs, including asynchronous communication channels: it adds two clauses to the OpenMP 3.0 task pragma for defining inputs and outputs [2]. The implementation of the ACOTES extensions to OpenMP 3.0 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, in which 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 follows these constraints, restricting the expressiveness of the language intentionally. The ACOTES programming model does not contain these limitations and, as we shall see in the following sections, 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.

2.2 Compiler Layer

The compiler layer provides support for translating and adapting the language constructs to the lower layers: it provides automatic transformations from sequential code to stream computations.

All the automatic parallelization techniques are based on the data dependence analysis information [11, 12, 7]. This static analysis determines the relations between memory accesses and allow the analysis of dependences between computations via memory accesses. This in turn allows task partitioning, data and computation privatization.

- Data dependences represent a relation between two tasks: in the case of flow dependences, the dependence relation is between a task that writes data and another task that is reading it. Figure 2 represents a regular data flow relation in which all the elements written by the producer are consumed. In Figure 3 only a part of the elements of the array are written, making a part of the read elements dependent on a previous producer. Figure 4 presents a more difficult dependence relation that cannot be determined at compile time: the consumer task is then considered dependent on the completion of the producer task.

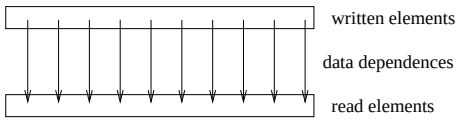


Figure 2: Regular flow data dependences

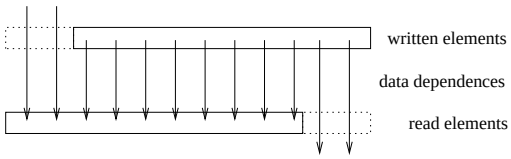


Figure 3: Shifted data dependences: a part of the dependences flow from an earlier producer and another part to a later consumer.

- Partitioning computation and data: among the compiler transformations that can generate tasks and communication channels is loop distribution,

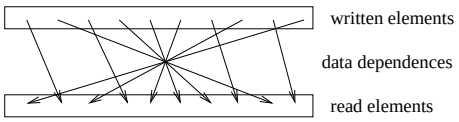


Figure 4: Irregular data dependences not known at compile time: the consumer has to wait for the last element produced for starting consuming (for example when written elements are $A(i)$ and read elements are accessed via an indirection: $A(B(i))$).

that can split a loop into several parallel loops that execute in pipelines. The maximal loop distribution algorithm is known as the Allen-Cocke-Kennedy algorithm for vector code generation [7]. It builds a dependence graph showing all dependences between statements, then walks in a topological sort order the statements of the loop and outputs vector versions of each statement. Cycles result in (minimal) serial loops. Array privatization or scalar expansion techniques are used for augmenting the parallelism generated by the loop distribution: these techniques allocate enough data for keeping track of all the array or scalar variable versions that the producer loop has written. This extra use of memory is needed to eliminate loop-carried dependences, making the loop distribution legal. In the following sections, we present a variation of the data privatization techniques: we limit the amount of duplicated memory to the size of a stream channel, reducing the overhead of privatization. Section 3.1 describes this dynamic privatization technique for loop distribution: it generates producer and consumer tasks, and the channels in between tasks replace shared memory accesses by FIFO channels.

- Coarse grain automatic parallelization is translating sequential code to parallel code by partitioning and distributing computation among several execution threads. Starting with version `GCC 4.3`, there is an infrastructure for automatically translating sequential code to parallel code using the `OpenMP` library. The support for this infrastructure has to be improved by parallelizing at a coarser grain outer loops, and also by improving the profitability function.

2.3 Runtime and Library Layer

The runtime and library layer provides an interface to the necessary services for streamization and parallelization. The runtime library is a target for the compiler layer translation of the language constructs into stream computation. It provides high-level communication and synchronization support that enable an efficient translation, as well as support for concurrent execution. The communication and synchronization functionality it provides further relies on the available OS and hardware support.

- `OpenMP` support in `GCC` is provided by the `GOMP` library. It currently implements the `OpenMP 2.5` standard, and an adaptation for `OpenMP 3.0` is available in a branch.
- Synchronization Arrays (`SA`) have been proposed in the framework of the *Decoupled Software Pipelining* [14] as a means of avoiding OS or spin-lock synchronization as well as shared memory communication between producer and consumer. It relies on hardware support to provide *produce* and *consume* operations. The `SA` will primarily behave as a queue, but it also allows for out-of-order execution as both producer and consumer use *dependence numbers* to identify the elements produced and consumed.

2.4 Platform Layer

The platform layer provides the low-level functionality used by the runtime library to implement its services.

The platform layer can also provide specific support for streamization. In the example of the `SA`s [14], the hardware support significantly increases performance as it replaces costly OS or spin-lock-based synchronization and avoids the cache pollution resulting from communication through shared memory. The authors acknowledge that, without this specific hardware support, they were unable to achieve any speedup with their *Decoupled Software Pipelining* technique.

In the next section, we describe the implementation of an automatic streamization framework for `GCC`. Then, we analyze several stream benchmarks that will allow us to tune the cost model of the automatic streamization.

3 Auto-Streamization in GCC

We propose here a framework for automatic generation of stream code in GCC, based on the techniques described in the previous section.

As we will see in the next paragraphs, we first need to provide a way to partition the computation into tasks that communicate in a way that is conducive to streamization. The second step is to provide suitable runtime support for stream communication and, finally, we need to generate calls to this runtime to enable both stream communication and concurrency.

3.1 Task Partitioning

The first step in the streamization process is to partition the computation into tasks that present a producer-consumer relationship. In other words, the tasks will have flow dependences between each other. If the dependence information is not computable at compile time, as for example in Figure 4, the compiler will not be able to streamize. In general, the producer and consumer originally communicate through a shared data structure, with the producer writing and the consumer reading. We replace this shared memory communication by stream operations.

In the current implementation, we use the loop distribution framework to partition a loop into such tasks. We focus on extracting tasks by distributing loops in which a loop-carried flow dependence prevents a trivial parallelization. The blocking nature of our stream implementation implicitly synchronizes the execution of the two tasks.

The loop distribution would split the following loop:

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

into the two following loops:

```
producer task | for (i=1; i<=N; i++)
              |   A[i] = ...;

consumer task | for (i=1; i<=N; i++)
              |   ... = ... A[i-1] ...;
```

3.2 Runtime Stream Support

To provide the functionality necessary for the streamization of the tasks we partitioned, we propose to extend the OpenMP library with the notion of streams. Streams are directional channels of communication that behave as a FIFO queue. The producer pushes elements into the stream while the consumer pops them.

We implemented this extension in libGOMP, GCC's OpenMP library. Our implementation is based on the stream implementation proposed in the ACOTES project [1] with some variations needed for reducing the amount of synchronizations. A stream is defined as a circular buffer to avoid excessive memory usage, but buffer's size could be dynamically resized if this appears to be a necessity. The buffer contains two sliding windows where the reads and writes to the buffer occur. These sliding windows are used for minimizing the amount of synchronization: sliding windows cannot overlap, such that elements read and written in these windows can be performed with no synchronization, allowing the synchronization to only happen when the windows are sliding. Furthermore the sliding windows can be aligned on cache boundaries, minimizing the number of cache misses: one or more full cache lines can be blocked on read or write mode, making them available exclusively in the caches of one processor before being evicted by an access request from another processor.

Figure 5 provides the current layout of the structure used for the streams. The implementation uses four pointers to track the positions of the two sliding windows and the positions of the written and read elements in these windows. The read_index and write_index fields are pointing to the elements of the buffers to be read or written. These two pointers are always pointing inside the sliding windows that start at read_buffer_index and write_buffer_index. The sliding windows have a length of local_buffer_size. The end of stream eos_p flag is mainly important in the case in which the number of elements that will be communicated is not known, even symbolically, to inform the consumer when the producer has finished.

This stream data structure is used by the interface for stream communication presented in Figure 6. It is important to note that the two access operations gomp_stream_push and gomp_stream_head are blocking operations. This means that if the stream buffer is

```
typedef struct gomp_stream {
  /* First element of the stream.  */
  unsigned read_index;

  /* First empty element of the stream.  */
  unsigned write_index;

  /* Size of sub-buffers for unsynchronized reads
     and writes.  */
  unsigned local_buffer_size;

  /* Index of the sliding reading window.  */
  unsigned read_buffer_index;

  /* Index of the sliding writing window.  */
  unsigned write_buffer_index;

  /* End of stream: true when producer has finished
     inserting elements.  */
  bool eos_p;

  /* Size in bytes of an element in the stream.  */
  size_t size;

  /* Number of bytes in the circular buffer. */
  unsigned capacity;

  /* Circular buffer.  */
  char *buffer;
} *gomp_stream;
```

Figure 5: Stream data structure

```
/* Returns a new stream of N * LOCAL_BUFFER_SIZE
   elements.  Each element is of size S bytes.  */
gomp_stream gomp_stream_create (size_t s, unsigned n);

/* Push element E in the stream S.  */
void gomp_stream_push (gomp_stream s, char *e);

/* Read the first element of the stream S.  */
char *gomp_stream_head (gomp_stream s);

/* Discard the first element of the stream S.  */
void gomp_stream_pop (gomp_stream s);

/* Check if the producer has finished inserting
   elements in the stream S.  */
bool gomp_stream_eos_p (gomp_stream s);

/* Set the end-of-stream flag for stream S.  */
void gomp_stream_set_eos (gomp_stream s);

/* Destroy the stream S.  */
void gomp_stream_destroy (gomp_stream s);

/* Push COUNT elements into the stream S, starting
   at the address START.  */
void gomp_stream_align_push (gomp_stream s,
                             char *start, int count);

/* Discard COUNT elements from the stream S.  */
void gomp_stream_align_pop (gomp_stream s, int count);
```

Figure 6: Stream interface

full, the `push` operation will block the producer until the consumer slides its read window. When the buffer is empty, the `head` operation will not return until the producer slides at least once the writing window. Producers use `gomp_stream_set_eos` to expose partially written write buffer when ending writing to a stream. To avoid deadlocks, the code generated using this interface must guard the use of these operations or have precise information on the tasks' behaviors.

We also provide two additional alignment functions, `gomp_stream_align_push` and `gomp_stream_align_pop` that allow us to ensure the elements pushed by the producer in the stream match the ones expected by the consumer as showed in Figures 2 and 3. Typically, this is necessary if, in the same iteration of a loop we distribute, the producer and consumer do not access the same element of an array.

Communication through streams rather than through shared memory also implies that the data is privatized, which constitutes an overhead, but could allow for more concurrency and cache locality on non-shared memory systems.

3.3 Code Generation

To parallelize the tasks we previously partitioned, we generate calls to the extended `OpenMP` library. We first enclose the tasks in `OpenMP` sections that will execute concurrently. Then we introduce calls to the appropriate functions from the stream extension to provide for communication and synchronization.

In the producer task, we generate a call to `gomp_stream_push` after the write that was at the origin of the flow dependence. Note that we cannot remove the write operation for the time being as we do not have a precise enough interprocedural analysis to decide if there are further uses of that memory location.

In the consumer task, we generate a call to `gomp_stream_head` instead of the read operation, then we call `gomp_stream_pop` to remove the element from the stream. Our decision to split this operation is to allow, in some cases, the removal of an unnecessary copy of the element from the stream to a temporary. This could be significant if the elements occupy a lot of space. For the consumer, we can safely remove the read operation.

A last step is to generate code to align the streams. As quite often the flow dependence along which we generate the stream is a loop-carried dependence, if we just replace the read and write operations by stream operations, the producer and consumer will not be temporally synchronized. In the example we proposed above, while the producer first pushes $A[1]$ into the stream, the consumer expects to first read $A[0]$. For this, we provide two alignment functions. To generate calls to the alignment functions `gomp_stream_align_push` and `gomp_stream_align_pop`, we need to know the number of elements we must align. The data dependence analysis provides us with this precise information in the form of the distance vector associated to this flow dependence. In the example we proposed above, the stream code generated by `GCC` is similar to what a programmer could write in `OpenMP` with calls to the `GOMP` streams, as shown in Figure 7.

```
gomp_stream s = gomp_stream_create (8, 16);
#pragma omp parallel sections num_threads (2)
{
#pragma omp section
  /* Producer task.  */
  {
    gomp_stream_align_push (s, A, 1);
    for (i=1; i<=N; i++) {
      elt e = ...;
      A[i] = e;
      gomp_stream_push (s, e);
    }
    gomp_stream_set_eos (s);
  }
#pragma omp section
  /* Consumer task.  */
  {
    for (i=1; i<=N; i++) {
      elt t = gomp_stream_head (s);
      gomp_stream_pop (s);
      ... = ... t ...;
    }
    gomp_stream_align_pop (s, 1);
    gomp_stream_destroy (s);
  }
}
```

Figure 7: Auto-streamization of the first example

As the stream operations have blocking semantics, *i.e.*, the producer waits until there is free space in the stream and the consumer waits for elements in the stream, the streams also provide synchronization between the producer and consumer tasks. For this reason, all further synchronization is superfluous and we can execute the two tasks concurrently.

As the stream code shows, the original read operation has been replaced with stream operations. However, the write operation remains and the stream operation is only added. This is due to the lack of a more precise interprocedural analysis, as we cannot know if subsequent reads to that memory location remain. This represents an important optimization opportunity.

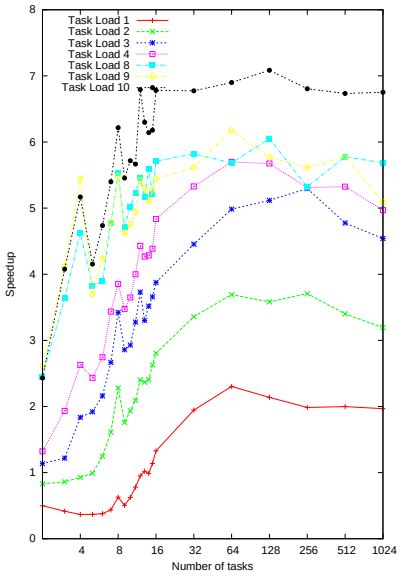
3.4 Stream Benchmarks

The evaluation of all the benchmarks presented in this paper is performed on an `AMD` Phenom 9550 machine with 4 cores, running at 2.2 GHz under Linux kernel 2.6.24, and the following characteristics of the memory hierarchy:

L1 cache line size	64 B
L1 cache	64 KB
L2 cache	512 KB per core
L3 cache	2 MB shared
RAM	4 GB

To evaluate the amount of computation that makes the stream computations faster than the sequential execution, we generated, starting from the loop kernels of the stream benchmarks, a set of benchmarks having more and more computations per iteration as follows: a loop iterates over all the elements of an array performing a computational task storing the result of the processed element back in the same array. The computational task in the vector scaling stream benchmark is a scalar multiplication. In our benchmarks, we used a basic task that contains more scalar computations: we used the computation of the Euclidean distance, $sqrt(a*a+b*b)$. To analyze the impact of the sequential computation load, we aggregated several computations in `Load1, ..., Load10` by repeating the basic task computation one to ten times. An array of 32 MB is processed successively by two or more filters. The code of the sequential case for two filters and the corresponding streamized code are presented in Figures 9 and 10. Figure 8 presents the speedup of these synthetic stream benchmarks. For a task load of one, the stream version is always slower than the sequential execution, showing that the amount of computation per task should be bigger. For a task load of two, the streamized version begins to be profitable starting with six concurrently executing threads. For a task load of more than two, the streamization is always beneficial.

The behaviour exhibited by these benchmarks proves that there is more interest in streamizing than just parallelization. The speedup achieved is superlinear in many cases. For two and four tasks with a task load of eight



Nb tasks	2	3	4	5	6	7	8	16	32	64
Load 1										
t_{Seq}	0.26	0.37	0.48	0.59	0.69	0.8	0.9	3.68	10.00	22.12
t_{Stream}	0.52	0.89	1.31	1.6	1.82	1.83	1.43	2.77	5.14	9.6
Speedup	0.5	0.42	0.37	0.37	0.38	0.44	0.63	1.33	1.94	2.3
Load 2										
t_{Seq}	0.73	1.06	1.4	1.99	2.82	3.64	4.47	11.11	24.28	50.73
t_{Stream}	0.88	1.23	1.51	2.01	2.26	2.26	1.96	3.96	7.23	13.74
Speedup	0.83	0.86	0.93	0.99	1.25	1.61	2.28	2.81	3.35	3.69
Load 3										
t_{Seq}	1.17	1.73	3.03	4.3	5.64	6.88	8.18	18.57	39.25	81.06
t_{Stream}	1.03	1.42	1.65	2.24	2.61	2.58	2.39	4.85	8.81	16.26
Speedup	1.14	1.22	1.84	1.92	2.16	2.67	3.42	3.83	4.45	4.98
Load 4										
t_{Seq}	1.64	3.17	4.89	6.67	8.4	10.17	11.87	25.63	53.82	109.19
t_{Stream}	1.24	1.64	1.86	2.74	3.06	2.96	3.08	5.5	10.10	19.16
Speedup	1.32	1.93	2.63	2.43	2.75	3.44	3.85	4.84	5.32	5.69
Load 8										
t_{Seq}	5.27	8.80	12.48	16.63	19.32	24.34	28.08	55.11	113.13	223.80
t_{Stream}	2.15	2.42	2.70	4.35	4.96	5.10	5.08	9.65	19.43	39.38
Speedup	2.45	3.63	4.62	3.82	3.89	4.77	5.52	5.71	5.82	5.68
Load 9										
t_{Seq}	6.04	10.14	14.60	17.86	22.54	26.38	31.66	65.81	128.08	270.48
t_{Stream}	2.31	2.46	2.68	4.83	5.32	5.52	5.77	12.10	22.82	43.78
Speedup	2.61	4.12	5.44	3.69	4.23	4.77	5.48	5.43	5.61	6.17
Load 10										
t_{Seq}	6.93	12.11	15.61	20.35	25.91	30.61	35.87	72.97	148.76	302.44
t_{Stream}	2.85	2.97	3.02	4.90	5.47	5.67	5.77	10.76	21.96	43.83
Speedup	2.43	4.07	5.16	4.15	4.73	5.39	6.21	6.78	6.77	6.90

Figure 8: Speedup of the streamization of stream benchmarks: the size of the sliding windows is set to 64B, matching the size of L1 cache lines, the circular buffer contains 64KB matching the size of L1 cache, and the amount of processed data is 32MB.

or more, the superlinear speedup is mostly due to an improved usage of caches and an increased size of cache available to the computation. As the tasks do not share a same L1 cache, they do not pollute each-other's caches. We also achieved superlinear speedup for all the cases where the speedup is above four as there are only four hardware threads available. Such is the case starting at a task load of three with more than 32 tasks or at a task load of four with 10 or more tasks. Similar superlinear behaviour can be observed for task loads of eight or above, for most task numbers, this is also a result of an improved cache behaviour. The drop in the speedup when adding one more task to four tasks is due to the fact that the processor that we are using has only four cores, and the fifth task has to be scheduled with another task on one of the four cores, slowing down the whole pipeline. The L3 cache is filled up by the stream buffers at about 32 tasks, making the speedup almost flat and even decreasing in some cases.

This analysis and other similar analyses will allow us to build a cost model that determines an appropriate task granularity that the compiler should use for deciding the split or aggregation of tasks in the automatic streamization pass. The ideas about the cost model are expanded in the next section together with potential improvements of several parts of the compiler that can improve the support for manual and automatic streamization in GCC.

```
int *A = (int *) malloc (N * sizeof (int));
for (i=1; i<=N; i++)
  A[i] = Load4 (A[i]);
for (i=1; i<=N; i++)
  A[i] = Load4 (A[i]);
```

Figure 9: Sequential stream filters

```
int *A = (int *) malloc (N * sizeof (int));
gomp_stream s = gomp_stream_create (4, 1000);
#pragma omp parallel sections num_threads (2)
{
#pragma omp section
  /* Producer task.  */
  {
    int i;
    for (i=1; i<=N; i++) {
      elt = Load4 (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);
      gomp_stream_pop (s);
      A[i] = Load4 (elt);
    }
    gomp_stream_destroy (s);
  }
}
```

Figure 10: Streamized stream filters

4 Future Work

As we have seen in the previous section, the support for streaming computations can be beneficial, but it also can slow down the execution with respect to the sequential execution due to the non-negligible cost of OpenMP parallelization and to the cost of synchronization of communications via the stream buffer. Programmers can already use the stream library support directly in conjunction with OpenMP parallelization, but our aim is to add, to the available tool-set of programmers, automatic and language support for streamization. For this, there remain several tasks to be completed as described in this section: on the language layer we advocate the adoption of ACOTES *in* and *out* clauses to the pragma task, which will make the use of streams easier when parallelizing with OpenMP. On the compiler layer, a major work on improving the precision of our static analyzers of data dependences in interprocedural mode has to be completed. On the runtime library layer we are investigating dynamic adaptation of the library to the execution context following feedback from the operating system and hardware counters.

4.1 Language Support for Streams

As mentioned in Section 2, the Brook language imposes a blocking task semantics. The ACOTES language extension is less restrictive: it can be used for both blocking tasks and non blocking tasks by selecting the number of sliding windows in the stream runtime library to be equal to 1, in which case the full stream has to be written and then read atomically. Because of this loss in expressiveness, we advocate the use of the ACOTES extensions to the OpenMP standard. The standardization of the *in* and *out* clauses to the pragma task and of the stream library interface is a natural step that does not necessitate an excessive effort. In the contrary, Brook's C extensions are non-trivial, as the C standard does not even define concurrency, making the Brook extensions quite hard to integrate in the current C standard. The implementation effort for supporting in GCC two additional clauses to the OpenMP pragma task is minimal, making the ACOTES extensions the best candidate for stream language support in GCC.

4.2 Improving GCC's Analyses

The most important analysis for streamization is the data dependence analysis. The precision of the data dependence analysis can improve the detection of parallel non-communicating tasks, and also allows detection of producer-consumer patterns. The major weakness of GCC's data dependence analysis is that it is limited to the code of a single procedure. This makes the analysis of code containing procedure calls impossible, reducing the automatic streamization opportunities. Interprocedural analyses (IPA) improve the precision of analyzed information by adding the information of the function call context. An IPA infrastructure has been integrated in GCC 4.1 and continues to be improved: the IPA mode was extended to the SSA representation for enabling constant optimization passes such as IPA constant propagation. To implement an interprocedural data dependence analysis, the value range propagation pass [13] has to be extended in IPA mode. Then, using this machinery, it is possible to gather and then propagate an abstract view of the reads and writes to memory that a procedure performs. This technique is known as array regions [10, 9]: reads and writes to an array are represented as constraint systems that are propagated in interprocedural mode.

4.3 Improving the Task Load

As we have seen in Section 3.4, the task load has a strong impact on the speedup that streamization can achieve. We need to be able to generate tasks that have sufficient load to attain a high level of speedup. One way this could be achieved is to transform the generated code in a manner quite similar to a loop unroll, aggregating in a single iteration multiple base iterations and performing stream operations on blocks of elements. We believe this may allow us to generate tasks that have a sufficient load and that would therefore present similar speedups as the higher task loads in the Figure 8.

4.4 Improving Stream Runtime

To uncover as much parallelism as possible, it might prove interesting to generate tasks at the finest granularity and provide a task fusion capability in the runtime system. This would require a slight modification of the way tasks are defined, a task then being the production of a single element or its consumption. As such, the runtime system could dynamically decide to fuse a producer task with a consumer task by invoking the consumer task immediately after the producer, without

stream operations. In the case in which the streamization of two tasks proves inefficient at runtime, their fusion should allow us to regain near-sequential performance.

We believe that specific support for NUMA architectures might prove profitable. As memory accesses to non-local memory become more expensive, a certain level of communication aggregation could reduce the increased latency of memory operations. Depending on the implementation of streams for NUMA architectures, the data can be stored either on the producer or on the consumer side. For example, if the data is stored on the consumer side, one possibility would be to buffer stream write operations on the producer side and only write aggregated blocks of elements to the consumer memory.

5 Conclusion

This paper presents several manual and automatic techniques for generating stream code from sequential code as well as some important performance considerations and optimization opportunities. These techniques are then applied to the implementation of an automatic streamization framework in GCC, followed by a discussion on the performance results that our implementation achieved on several stream benchmarks. As we have seen, the profitability is strongly connected to the amount of work we can isolate per task as well as to the number of tasks. We propose a maximal partitioning of tasks in the compiler, leaving the runtime library to decide on the fusion of tasks following the amount of computation per task. We also proposed improvements to the current GCC static analyses infrastructure for more precise data dependence information. This information is crucial to the translation of sequential code to SIMD and MIMD.

6 Acknowledgments

We would like to thank Razya Ladelsky, Uzi Shvadron, and Ayal Zaks from IBM Haifa, and 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] The StreamIt language. http://www.cag.lcs.mit.edu/streamit/.

[4] The Brook Language. http://graphics.stanford.edu/projects/brookgpu/lang.html.

[5] The CUDA Language. http://www.nvidia.com/object/cuda_home.html.

[6] The OpenMP Standard. http://www.openmp.org/.

[7] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan and Kaufman, 2002.

[8] E. Ayguadé, N. Copty, A. Duran, J. Hoe-flinger, Y. Lin, F. Massaioli, E. Su, P. Unnikrishnan, and G. Zhang. A proposal for task parallelism in OpenMP. In *3rd International Workshop on OpenMP (IWOMP)*, June 2007.

[9] B. Creusillet. *Array Region Analyses and Applications*. PhD thesis, Dec. 1996.

[10] B. Creusillet and F. Irigoin. Interprocedural array region analyses. *IJPP*, 24(6):513–546, Dec. 1996.

[11] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, 1967.

[12] L. Lamport. The parallel execution of do loops. *Commun. ACM*, 17(2):83–93, 1974.

[13] D. Novillo. A propagation engine for GCC. In *GCC Developers Summit*, pages 175–185, 2005. http://www.gccsummit.org/2005.

[14] R. Rangan, N. Vachharajani, M. Vachharajani, and D. August. Decoupled software pipelining with the synchronization array. In *PACT*, Sept. 2004.