Automatic Streamization in GCC

Antoniu Pop
MINES ParisTech, Centre de Recherche en Informatique, Mathématiques et Systèmes
antoniu.pop@mines-paristech.fr

Sebastian Pop Jan Sjödin
Open Source Compiler Engineering, Advanced Micro Devices, Austin, Texas
firstname.lastname@amd.com

Abstract

Multi-cores and multi-processors became ubiquitous during the last few years, and the trend is to increase the number of simple, power-efficient, and slower cores per chip. One of the results is that the performance of single-threaded applications did not significantly improve, or even declined, on new processors, which heightened the interest in compiler automatic parallelization techniques.

Our objective is to develop a framework in GCC to transform loops into pipelines of concurrent tasks using streams to communicate and synchronize. This transformation can either rely on user hints (pragmas) or on static analysis of control and data dependences. Our focus is to fully automate this transformation, which requires an integration with the Graphite polyhedral loop optimization framework and will lead to the development of runtime optimizations based on this polyhedral representation. This approach is complementary with the existing parallelization passes. This paper presents the steamization technique, the optimizations it enables, the interaction with other optimizations, and the necessary extensions to Graphite/PCP (polyhedral compilation package) for its integration.

1 Introduction

Increasing clock speed as well as micro-architectural and compiler advances have allowed steadily improving performance of single-threaded programs for many years. However, this does not seem to be the case any longer. The excessive design complexity and power constraints of large monolithic processors made this paradigm unsustainable and ultimately forced the industry to develop chip-multiprocessor (CMP) architectures, in which multiple processor cores are tiled on a single chip. As the number of transistors per chip continues to grow exponentially, the current trend is towards providing an increasing number of simpler, more power-efficient, and slower cores per chip. The performance of single-threaded applications is therefore expected to stagnate or even decline with new generations of processors. The applications need to exploit the multiple hardware execution threads available on these architectures to improve performance.

Besides the parallel applications, already covered by GCC's automatic parallelization pass, parallelization often requires enabling transformations such as privatization, which allows removal of false dependences (write after read and write after write). However, this transformation often comes with a very high cost both in the amount of memory required and in execution time. The common privatization technique in which dynamic single assignment properties are achieved may even be impossible in some cases because the hardware resources are finite. Streamization allows reduction of the amount of memory necessary for privatization (compression after the expansion of memory). Though this comes with a reduction of the amount of parallelism exploitable, it also brings some benefits to the way communication through shared memory happens. The main objective of streamization is to optimize privatization.

Another objective of streamization is to achieve pipeline parallelism, which is complementary with data parallelism. It consists of generating pipelines of concurrent tasks communicating and synchronized through streams [12], which behave as blocking FIFO queues. We will elaborate on the specificities of our stream implementation in Section 2. One of the salient problems for parallelization on current architectures is to get data at the right place, at the right time. This problem becomes increasingly acute with each new generation of processors as the memory wall builds up. Despite the fact that the clock speed of new processors is no longer increasing, more processing units become available, sharing the same limited resources. The number of CPU cycles available on CMPs per unit of time is increasing faster than the memory bandwidth because a single data bus has to feed more processing units. To fully exploit the available resources, applications need higher arithmetic intensity. Also, cache effects (in particular in the case of multiple hardware threads sharing cache lines) make scheduling decisions ever more complicated. We will try to show that pipeline parallelism can improve the behaviour of parallelized applications in that regard.

The paper starts by presenting the details of the stream communication library in Section 2. Section 3 presents the application of streamization to pipeline parallelism as well as experimental results of this technique on a kernel extracted from the GNU Radio project. Section 4 shows how streamization can be used to optimize privatization. In Section 5, we describe the techniques we use to ensure that streamization does not inhibit other important optimizations such as vectorization or automatic parallelization. Section 6 presents the integration of streamization with Graphite. Finally, in Section 7 we evaluate the benefits of our approach with regards to previous contributions.

2 Privatization and Streams

Privatization is a technique used for eliminating false (or storage-related) data dependences (write after read and write after write dependences), in order to expose parallelism or enable program transformations. This technique consists in duplicating some memory area, with various levels of duplication. Depending on the context, it means making some memory area private (or local) to a thread, or to a point in an iteration domain, or even reaching a dynamic single assignment (DSA) property.

Let us consider the following example:

```
int a;
for (i = 0; i < N; ++i) {
  a = ...;
  ... = ... a ...;
}
```

If we wish to parallelize this loop on P threads, it is sufficient to make a private to each thread. This means that P copies of a are necessary and we can distribute the iterations of the original loop among the threads. One possible solution, with the outermost loop fully parallel, is:

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

Here the resulting code still has false dependences (not DSA), but they do not span across multiple outermost loop's iterations. If our objective was to distribute the original loop, then we need to make more copies of a, one for each point of the iteration domain:

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

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

There are no more false dependences (DSA code) and the iterations of the resulting loops are parallel. This type of privatization is also referred to as scalar expansion (or array expansion if a was of higher dimension).

In the rest of the paper, we will generally refer to scalar or array expansion as memory expansion, while privatization will be used as the general notion of making at least enough copies to enable a transformation. When more copies than necessary have been created (*e.g.*, by memory expansion), reducing the number of copies will be referred to as memory compression.

2.1 Stream Communication Library

The stream communication library provides a simple interface for using streams, which are blocking FIFO queues, for communicating between two threads. The communication is unidirectional and ordered. The interface provides simple access operations to the elements in the stream, a push operation for inserting one element at the end of the stream, and a pop operation for removing the first element in the stream. Some more complex operations are also implemented for optimization purposes.

The notion of stream is not new and often has different accepted semantics. In some cases, streams behave as bags of elements in which no order is enforced on the access operations. This is notably the case in accelerator-oriented streaming languages (*e.g.*, Brook or CUDA) because the accelerator often does not provide for synchronization.

This library is not meant for programmers' use, but as a target for code generation, in particular in the *Graphite codegen* pass[1]. In this case, streams are used to replace arrays and thus compress memory (counterpart of memory expansion or privatization), replacing the strong single assignment property by a relaxed single assignment in which memory is reused after a synchronization point that ensures the overwritten data is no longer useful. We will elaborate on this use of stream communication in Section 4.

2.2 Simplified Stream Semantic

To make the use of streams straightforward in optimization frameworks such as the polyhedral compilation package (PCP), we need to provide a simple approximation of the stream semantic. Though this approximation does not correspond to the underlying implementation, it should be restrictive enough to ensure the correctness of the generated code.

In most cases, streams will simply be considered as infinite arrays in which elements are only written and read once, in sequential order, like a FIFO queue.

As data is duplicated and, in this simplified semantic, each element is assigned only once, we will consider privatization through streams to provide DSA.

Such a semantic clearly goes against the objective of reducing the memory used for privatization and is not implementable with finite hardware resources. We need to impose some restrictions on the communication patterns to allow reusing memory.

2.3 Implementation and Interface

The implementation of streams is in the form of directional channels of communication that behave as blocking FIFO queues. The producer enqueues elements into the stream and the consumer dequeues them. Streams

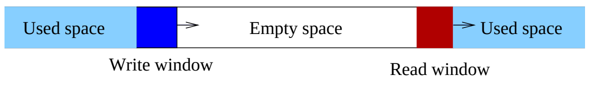

Figure 1: Sliding windows in a stream buffer.

are implemented as circular buffers to avoid excessive memory usage, but the buffer can be dynamically resized if this appears necessary. The blocking behavior means that the queuing operation blocks when the buffer is full and the dequeuing operation blocks when it is empty.

Stream communication serves two purposes: first, it privatizes the data, thus removing any output or anti-dependences; second, it relaxes the flow dependences as it decouples the producer from the consumer. Because 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.

The implementation of streams without *ad hoc* hardware support presents two principal sources of overhead. The first is the need to synchronize read and write operations, which can be very expensive. The second problem comes from bad cache behaviour due to false sharing when the producer and consumer access elements that are close together. Because the stream elements are stored in a circular buffer, this happens when the buffer is almost full or almost empty.

Both of the sources of overhead can be almost completely mitigated by adjusting the granularity of communication. The synchronization overhead can be reduced by aggregating multiple elements in blocks and only synchronizing the accesses to blocks of elements. The cache degradation can be avoided by preventing the producer and the consumer from accessing elements in the same cache line.

To increase the granularity of the communication, we introduce sliding windows (see Figure 1), in which the reads and writes to the buffer occur. These sliding windows are used for reducing the amount of synchronization, which is required only when the windows are sliding. The windows can also be aligned on cache boundaries to avoid false sharing. One or more cache lines are reserved for writing/reading to/from the stream.

[1]For details, see the "Design of Graphite and the Polyhedral Compilation Package" paper in the GCC Summit 2009 proceedings.

The interface of the streaming library provides both basic access functions, like `push` and `pop`, and more efficiency-oriented functions that avoid unnecessary copies and library calls. The basic interface provides the following simple access functions:

```
void push (stream, element);
element pop (stream);
bool end_of_stream (stream);
```

The `push` and `pop` functions are the usual access functions to FIFOs, but it should be noted that both represent copies. The `end_of_stream` function checks whether the producer has finished working and the stream is empty. It should be called whenever it is not possible to test for termination in other ways. This is necessary, for example, when the producer loop is under dynamic control, so that – even at runtime – it is impossible to know the number of iterations until the loop finishes. In most examples in this paper, the producer and consumer iterate on the same domain, so there is no need for this check, but examples can be found in Sections 3.1 and 4.3.

The remaining access functions allow direct access in the stream buffer, one window at a time. To reduce overhead, a task can request an empty window from the stream and store the elements directly, using `get_tail_window`. Once the window has been filled, `commit_window` makes it available for reading by the consumer task. The same mechanism is available for the consumer. An illustration of the application of this technique can be found in Section 5.

```
element *get_tail_window (stream);
void commit_window (stream);

element *get_head_window (stream);
void pop_window (stream);
```

These operations not only reduce the runtime overhead, they also avoid useless copies. They would also allow a seamless integration with software transactional memory [9] to enable speculative execution of consumer threads.

3 Exploiting Pipeline Parallelism with Loop Streamization

Loop streamization is a program transformation that enables pipeline parallelism in sequential programs. As with other similar techniques that we further discuss in Section 7, it relies on memory expansion (privatization) and synchronization. This technique is primarily based on loop distribution and software pipelining.

Our objective is to automatically enable and exploit parallelism in sequential programs while avoiding non-scalable expansion schemes. As we will see, streamization will allow us to explore the entire space of memory expansion, ranging from the original sequential code to the highest level of parallelism with full privatization (which is often non-realistic). The choice of the amount of memory duplication, and therefore of the amount of parallelism enabled, can be both static and/or dynamic.

3.1 Streamizing `while` Loops

The first step in the streamization process is to partition the computation into tasks that present a producer-consumer relationship. In the general case, tasks will have flow dependences in between each other; otherwise, they are only bound by control dependences. The producer and consumer originally communicate through a shared data structure, in which the producer writes and the consumer reads. We replace this shared memory communication by stream communication. The blocking nature of our stream implementation implicitly synchronizes the execution of the producer and consumer tasks.

The static analysis involved in partitioning a loop into tasks is identical to that of loop distribution [11, 6]. After building the program dependence graph [10], the strongly connected components are coalesced. The nodes of the resulting directed acyclic graph can be partitioned with, for example, an iterated minimum cut algorithm because each cut edge will represent inter-thread communication. Another option is to try to statically balance the load of each partition using a sparsest cut algorithm with weighted vertices.

As an illustration of the task partitioning, consider the following `while` loop:

```
S1:  while (data = read (input))
       {
S2:      tmp = process (data);
S3:      write (tmp, output);
       }
```

This type of loop actually represents a very common case. The `read` operation can be thought of as reading from a file in applications like video decoding or getting the next element in a linked data structure (list,

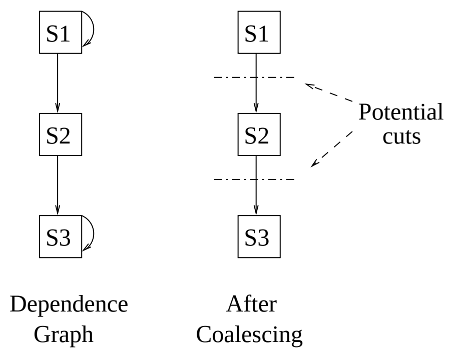

Figure 2: Dependence graph and the DAG obtained after coalescing strongly connected components.

tree ...). In most cases, this operation cannot be executed concurrently because it updates the state of the `input` parameter. The same remarks hold for the `write` operation and its `output` parameter.

For this code, the dependence graph is presented in Figure 2. In this simple case, the coalescing of the strongly connected components only removes the self-cycles on S1 and S3, exposing the potential cuts.

The parallelization of this loop can be achieved either as a doacross schedule or by pipelining. The doacross parallelization schedules iterations of the loop on different threads and introduces synchronization for each cross-iteration dependence (we do not take into account output and anti-dependences because they can be eliminated by privatization [8]). The pipelining approach will schedule each strongly connected component in the dependence graph on a different thread and synchronize each inter-thread flow dependence [14]. This synchronization being implicit in streams, we would get the following streamized pipeline:

```
while (data = read (input)) {
  push (S_data, data);
}

while (! end_of_stream (S_data)) {
  tmp = process (pop (S_data));
  push (S_tmp, tmp);
}

while (! end_of_stream (S_tmp)) {
  write (pop (S_tmp), output);
}
```

To understand the reason why pipelining is more efficient than other approaches, we show in Figure 3 the doacross and pipelined schedules.

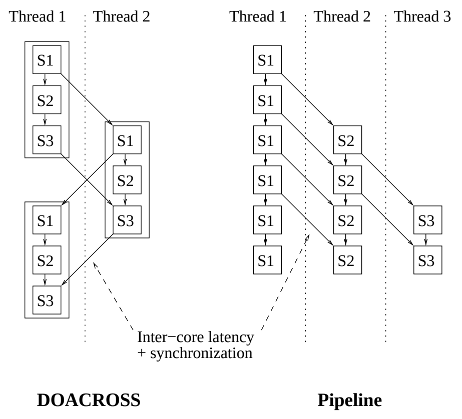

Figure 3: Doacross and pipeline schedules for the `while` loop.

If we compare the execution traces obtained by the two techniques, it is easy to see that pipelining will be more efficient because it shortens the critical path. The insight here is that the inter-core synchronization plus the communication of the data will introduce a high latency. This will be problematic if such latency is allowed on the critical path. In the doacross schedule, the dependence that is enforced across threads is a cross-iteration dependence, which means that the underling memory location cannot be privatized. The synchronization and communication overhead must be paid at least once for each iteration of the loop. However, in the pipeline schedule, we keep such loop-carried dependences on the same thread (this comes from the fact that each strongly connected component of the dependence graph is scheduled on a single thread). The remaining dependences will still have the same overhead, but privatization will better tolerate (or hide) the latency.

3.2 Dynamic Loop Fusion

Another way to look at loop streamization is to consider that it is like a halfway position between distributed and fused loops. Depending on the amount of memory used in the stream buffer for duplication, the producer and consumer loops can be more or less coupled. Consider the following streamized loop:

```
stream S;
for (i = 0; i < N; ++i) {
  push (S, ...);
}
for (i = 0; i < N; ++i) {
  ... = ... pop (S) ...;
}
```

If the stream S only allows storage of a single element at a time, the possible execution schedule of the two loops will be identical to that of the fused loops:

```
stream S;
for (i = 0; i < N; ++i) {
  push (S, ...);
  ... = ... pop (S) ...;
}
```

Such a fusion (as well as the replacement of the stream by a scalar) is always possible because the producer and consumer necessarily traverse the same iteration domain. On the other hand, if the stream allows an infinite number of elements to be stored at a time (or, in this case, at least N elements), then the possible schedules can be as decoupled as distributed loops.

3.3 Experimental Evaluation

We evaluate the potential of the streamization technique on a kernel extracted from the GNU Radio project [5]. 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 4. We will show, as a motivation for the optimizations under development, what can be gained from streamizing the code. We write the code as it would be generated by an optimizing compiler, with no additional manual optimizations. The implementation of the streaming library takes advantage of the memory hierarchy by aggregating communication in reading/writing windows. These windows should at least be of the size of an L1 cache line, which reduces false sharing and improves performance [12].

The OpenMP annotations we use constitute a minor extension to the OpenMP3.0 standard. We only introduce the association of the `lastprivate` clause on `task` constructs. The presence of this clause means that the corresponding task produces a value that needs to be propagated to the enclosing context, so anny subsequent task will see this value. In other words, variables marked with `lastprivate` are produced by the task and variables marked with `firstprivate` are consumed. This knowledge allows to build pipelines of producer/consumer tasks.

The evaluation of this benchmark is performed using a modified version of GCC4.4 available in the streamOMP branch. The experimental results are presented on Figure 5.

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 framework 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.

4 Optimizing Privatization with Streams

To expose parallelism in a sequential program or to enable loop transformations, it is often necessary to remove false dependences by privatizing the memory locations involved in these dependences. Though privatization enables some optimizations, it can be excessively expensive, both in terms of memory requirements and execution time. We will first show how streamization can improve the memory requirements of privatization for the purpose of enabling an optimization.

4.1 Reducing the Memory Footprint

Privatization through memory duplication has the benefit of exposing the maximum amount of parallelism, but it also is the most expensive technique in terms of memory usage. To avoid an excessive increase in the memory footprint when there are no loop-carried dependences, a common technique consists of only making one copy per concurrent thread. However, this is not always possible.

Consider the following example:

```
int a;

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

```
#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);
          }
      }
    }
  }
```

Figure 4: Kernel extracted and adapted from the GNU Radio project [5] with OpenMP annotations. The lastprivate clauses on tasks enable streamization.

In this case, the loops are sequential and the array B is not traversed in the right order. Loop interchange is necessary to improve performance, but it is not possible in this imperfect loop nest. The first step is to expand the scalar a and distribute the outermost loop, then we can interchange the loops to get the following code:

```
int A[N];

for (i = 0; i < N; ++i)
```

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

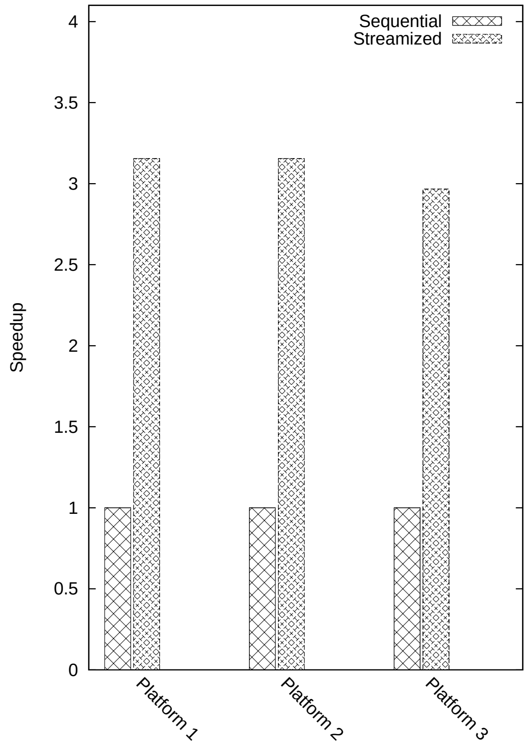

Figure 5: Speedups to sequential execution obtained on the GNU Radio kernel presented on Figure 4.

```
  A[i] = ...;

for (j = 0; j < M; ++j)
  for (i = 0; i < N; ++i)
    A[i] += B[j][i];

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

Now the array traversal is correct and the innermost loop is parallel. If we want to parallelize the outermost loop, we cannot interchange (or we would just go back to an inefficient traversal of B), so we need to further privatize the array A. However, the expansion of array A does not parallelize the outermost loop (see below), and loop skewing will not allow for a higher granularity of parallelism than the innermost loop parallelization.

```
int A[M][N];

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

for (j = 0; j < M; ++j)
  for (i = 0; i < N; ++i)
    A[j+1][i] = A[j][i] + B[j][i];

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

If, instead of a scalar expansion followed by an array expansion, we use stream privatization, we can both reduce the amount of memory used and relax the synchronization. We obtain the following code:

```
stream S[M];

for (i = 0; i < N; ++i)
  push (S[0], ...);

for (j = 0; j < M; ++j)
  for (i = 0; i < N; ++i) {
    tmp = pop (S[j]) + B[j][i];
    push (S[j+1], tmp);
  }

for (i = 0; i < N; ++i)
  ... = ... pop (S[M]) ...;
```

One interesting thing to note is that, while skewing allows to parallelize along Lamport hyperplans, streamization allows a much more relaxed wavefront parallelization schedule, as shown on Figure 6.

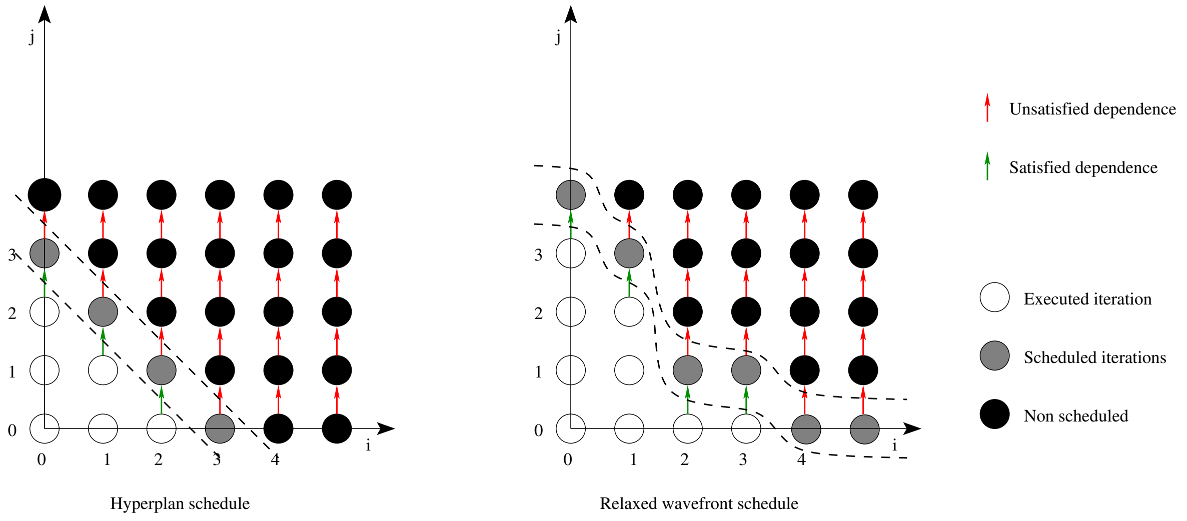

Figure 6: Lamport hyperplan schedule of the loop nest and relaxed scheduled of the streamized loop nest.

The amount of memory used to privatize the scalar `a` is also reduced by streamization. As we noted in Section 2.3, a stream can be made to use as little as a single copy of the privatized memory area (the scalar `a` here) or any arbitrary integer σ (the size of the stream) multiple of this amount. In the original code, `a` occupies $O(1)$ memory space. After the scalar expansion, the array `A` used for privatization uses $O(N)$ and then $O(M*N)$ after array expansion. The streamized version of the code uses $O(\sigma*M)$ space, which can be significantly lower than its non-streamized counterpart.

4.2 Memory Duplication vs. Parallelism

A further optimization is possible on the previous example if we only make one stream copy per thread. Consider that the outermost loop will be executed in parallel over `P` threads. Then it is not necessary to use memory expansion to privatize the streams used to communicate between the different iterations of the outermost loop. The following code would result from this memory compression:

```
stream S[P];

for (i = 0; i < N; ++i)
  push (S[0], ...);

for (k = 0; k < P; ++k)
  for (j = k; j < M; j += P)
    for (i = 0; i < N; ++i) {
      tmp = pop (S[k]) + B[j][i];
      push (S[(k+1)%P], tmp);
    }
for (i = 0; i < N; ++i)
  ... = ... pop (S[0]) ...;
```

In this final form, the privatization of `a` only requires $O(\sigma*P)$ copies. This is especially interesting because we can control the amount of parallelism available by increasing the amount of memory duplication at the finest granularity. If we increase the size of the sterams σ, we can reduce the coupling between the concurrent threads, while `P` directly controls the amount of parallelism available. This allows exploration of the full space of memory expansion, from scalar to multidimensional array expansion (if we make streams of size `N` and `P = M`).

One interesting benefit of our approach is to allow us to very precisely choose the tradeoff between the amount of parallelism made available and the amount of memory used in the privatization process.

4.3 Postamble to Privatization

In most privatization techniques, there is a preamble (duplicating the memory) and a postamble (storing in the original memory the values that would have been generated in the sequential computation). This postamble is necessary if the memory is read afterwards or if it is not possible to determine whether it is still needed. This last step can be fairly expensive and complicated in some cases. However, the use of streamization based privatization makes this trivial. As streamization requires the sequentialization of the computation in the pipeline, the last computed value for each memory location corresponds to the last value stored in the stream used for privatizing that location. It is therefore enough just to store back to the original memory location the last value of the stream, which is an O(1) operation.

For example, memory expansion in the following code requires the last computed value of `a` to be kept for the use statement after the loop:

```
int a;
for (i = 0; i < N; ++i) {
  if (condition (i))
  {
    a = ...;
    ... = ... a ...;
  }
}
use (a);
```

If we privatize, we get the following:

```
int A[N];
int a;
for (i = 0; i < N; ++i) {
  if (condition (i))
  {
    A[i] = ...;
    ... = ... A[i] ...;
  }
}
a = postamble (A);
use (a);
}
```

where `postamble` returns the element in array `A` that was last assigned. If the loop was executed in parallel, then it is necessary to keep track of all stores to the array and find the maximum on the indices in the array where a store occured. Though this operation is parallelizable, the operation requires $O(N)$ steps.

If the loop was streamized instead, it is sufficient to store in `a` the last element in the stream:

```
int a;
stream S;
for (i = 0; i < N; ++i) {
  if (condition (i))
  {
    push (S, ...);
  }
}
while (! end_of_stream (S)) {
  ... = ... pop (S) ...;
}
a = last_element (S);
use (a);
}
```

5 Interaction with GCCOptimization Passes

It is important to ensure that the streamization pass does not inhibit, or hinder the applicability of, other optimization passes in GCC.

Our objective is to ensure that if, for example, a loop is vectorizable prior to streamization, then the streamized loop also benefits from vectorization.

To achieve this, we introduced the window operations described in Section 2.3. Using these operations, we achieve some form of loop blocking in which the inner loop will present the same structure as the original loop, and which is therefore equally vectorizable.

Without entering into the implementation details, the stream implementation relies heavily on aggregation of multiple elements in windows (see [12] for details). This has multiple advantages, in particular for cache behavior and synchronization overhead reduction. In the following code, the streamization process does not access the stream element-wise, as we used in the other examples, but by blocks of `window_size` elements at a time. This allows the innermost loop, which iterates over each one of these blocks, to have a regular behavior conducive to vectorization.

We used this example in previous sections:

```
int a;
for (i = 0; i < N; ++i) {
  a = ...;
  ... = ... a ...;
}
```

instead of the following streamized loop in which no further optimization is possible due to the access function calls:

```
stream S;
for (i = 0; i < N; ++i) {
  push (S, ...);
}
for (i = 0; i < N; ++i) {
  ... = ... pop (S) ...;
}
```

We will have:

```
stream S;
int *w1, *w2;

for (i = 0; i < N; i += window_size) {
  w1 = get_tail_window (S);
  for (j = 0; j < window_size; ++j) {
    w1[j] = ...;
  }
  commit_window (S);
}
for (i = 0; i < N; i += window_size) {
  w2 = get_head_window (S);
  for (j = 0; j < window_size; ++j) {
    ... = ... w2[j] ...;
  }
  pop_window (S);
}
```

The outer loop handles synchronization and communication, while the nested loop is very similar to what we could have obtained by expanding the scalar `a` and distributing the loop, as we did in Section 2. The important parameter will be the window size, which will determine profitability: a small sliding window would create too much synchronization, while a too-large window would make the processed data not fit in the caches.

In the fully dynamic case, the runtime may determine the size of windows, but at the expense of having to make the vectorization decisions at runtime and using a vector and scalar version of the computation task.

In the static transform case, a part of the synchronization is transformed into static control using loop blocking: this enables vectorization at compile time and eliminates some runtime checks. The window size chosen at compile time may not be the best because the memory communication and synchronization costs are less precise at compile time.

The stream runtime dynamically performs loop fusion and loop blocking, operations that may be performed at a lesser cost at static time by the Graphite framework, but with greater uncertainty on the dynamic costs.

6 Integration with Graphite/PCP

The data flow analysis of memory accesses is available in the polyhedral representation GPOLY of the PCP infrastructure. Some of the transformations performed by GPOLY involve data privatization that can be optimized using streams. GPOLY tags some of the arrays that have been used for privatization as streams when the data flows through the array as through a FIFO. PCP then annotates the dimensions of the arrays that can be compressed into streams, and the code generation produces the streamized code without further analyses.

6.1 Data Flow Analysis for Streamization

A stream can be used when the memory communication between a consumer and a producer has the following properties:

- Source and target iteration domains are equal: the number of points and the iteration order over these points are identical; and,
- Data dependences between producer and consumer are regular: the consumer must read the data in the same order it was produced.

Under these conditions the dimensions that can be contracted into a stream are marked with an annotation by PCP.

6.2 A Stream Extension to PCP

In PCP, the streams are represented as arrays with one of their dimensions annotated with the `stream` flag:

```
streamType <- array(10, 100 | stream(1))
```

This example defines the type of an array of 10 by 100 elements, in which the second dimension containing 100 elements could be compressed using a stream: the `stream` annotation can be ignored, in which case a full size array has to be generated.

6.3 Stream code generation from PCP

Streams are created in the Graphite code generation. The stream annotation allows the code generation to transform an array into a stream without further data flow analysis. There is no need to communicate the end of generated data between the producer and the consumer because the iteration domains in which they occur are identical. This, and the fact that a stream array is defined and used only once, allows the code generator to always insert the initialization and the finalization of the streams before the loop nest of the producer and after the loop nest of the consumer.

7 Related Work

Stream programming has recently attracted a lot of attention as an alternative to other forms of parallel programming that offers 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 [2] 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 [3] 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 [4] 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 such as a new task pragma with clauses for defining inputs and outputs [7]. 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 because 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 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 [13] 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.

The technique that is closest to our approach is the decoupled software pipelining (DSWP) [14] proposed by Rangan *et al*. The authors extract parallelism by building the program dependence graph, then isolating in separate threads the strongly connected components of the graph. They rely on hardware support in the form of synchronization arrays and evaluate their code on a simulator. They recognize that, without hardware support, their technique only results in slowdowns. The static analysis used in this framework is unable to handle cases other than loop distribution.

8 Conclusion

We presented some motivating factors for the extension of the Graphite and PCP optimization infrastructures with streamization. Streamization allows exploiting pipeline parallelism in otherwise sequential loops and we have showed that its application to a GNU Radio kernel results in interesting speedups. This technique also allows reducing the amount of memory used for privatization and to finely explore the tradeoff between parallelism and memory expansion.

The paper details the interactions of streamization with other GCC optimizations and suggests an extension to PCP for integration with Graphite.

References

[1] ACOTES: Advanced Compiler Technologies for Embedded Streaming. `http://www.hitech-projects.com/euprojects/ACOTES/`.

[2] The StreamIt language. `http://www.cag.lcs.mit.edu/streamit/`.

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

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

[5] The GNU Radio project. `http://www.gnu.org/software/gnuradio/`.

[6] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[7] P. Carpenter, D. Ródenas, X. Martorell, A. Ramírez, and E. Ayguadé. A streaming machine description and programming model. In S. Vassiliadis, M. Berekovic, and T. D. Hämäläinen, editors, *SAMOS*, volume 4599 of *Lecture Notes in Computer Science*, pages 107–116. Springer, 2007.

[8] R. G. Cytron. *Compile-time scheduling and optimization for asynchronous machines*. PhD thesis, Champaign, IL, USA, 1984.

[9] U. Drepper. Parallel programming with transactional memory. *Queue*, 6(5):38–45, 2008.

[10] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[11] K. Kennedy and K. S. McKinley. Loop distribution with arbitrary control flow. In *Supercomputing '90: Proceedings of the 1990 conference on Supercomputing*, pages 407–416, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[12] 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.

[13] 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.

[14] R. Rangan, N. Vachharajani, M. Vachharajani, and D. August. Decoupled software pipelining with the synchronization array. In *13th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, Sept. 2004.