Task Fusion in Data-Flow Task-Parallel Applications

Igor Wodiany*, Antoniu Pop*

* *Department of Computer Science, The University of Manchester, United Kingdom*

ABSTRACT

OpenStream is an expressive data-flow task-parallel programming model implemented as a streaming extension to OpenMP. It allows the development of task-parallel applications with data-flow dependencies and dynamic tasks graphs, where tasks communicate using private data streams. Despite existing efficient scheduling techniques that achieve high data locality, the performance of OpenStream applications still strongly depends on the program partitioning done by the programmer. To relieve the developer of this responsibility and to achieve high performance, the development of automated compile-time and runtime approaches to task fusion has long seemed a promising solution lacking an efficient implementation. Although previous work addresses these issues using the polyhedral model, many problems still remain unsolved. In this poster we give an overview of task fusion in OpenStream.

KEYWORDS: Data-flow task-parallelism; OpenStream; Compiler optimizations; Task fusion

1 Introduction

Many data-flow task-parallel programming models have been proposed. Languages such as OpenMP[2] and OmpSs [DAB+11] allow the development of applications composed of dependent tasks that communicate through shared memory. Habanero-Java (HJ) [CZSS11] enables development of large scale applications with dynamic tasks graphs and fine-grained synchronization using *phasers*. Finally OpenStream [PC13] offers some of the benefits of previous approaches, but also introduces data privatization as a part of the model.

OpenStream, an expressive data-flow task-parallel model, is implemented as a streaming extension to OpenMP. It allows the development of task-parallel applications with dynamic tasks graphs that express producer-consumer relationships between tasks. Tasks synchronize and communicate using data streams. Conceptually, streams are infinite in size and privatized data is accessed by tasks using sliding windows. The compiler and runtime enable the execution of such applications on shared memory architectures. An example of an OpenStream task specification can be seen in Figure 1.

The OpenStream runtime enables efficient scheduling techniques, that exploit data privatization and known producer-consumer relationships, to achieve high data locality [DPH+16]. Despite that, achieving high performance requires the programmer to correctly partition ap-

[1]E-mail: {igor.wodiany, antoniu.pop}@manchester.ac.uk

[2]`https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf`

```
#pragma omp task input(sright[i0 - 1] >> left_in[1],
                       scenter[i0] >> center_in[1],
                       sleft[i0 + 1] >> right_in[1])
                output(sleft[i1] << left_out[1],
                       scenter[i1] << center_out[1],
                       sright[i1] << right_out[1])
{
  double result = compute(left_in[0], center_in[0],
                          right_in[0]);
  left_out[0] = result;
  center_out[0] = result;
  right_out[0] = result;
}
```

(a) OpenStream task computing one element of the 1d stencil computation per task.

```
#pragma omp task input(sright[i0 - 1] >> left_in[1],
                       scenter[i0] >> center_in[block_size],
                       sleft[i0 + 1] >> right_in[1])
                output(sleft[i1] << left_out[1],
                       scenter[i1] << center_out[block_size],
                       sright[i1] << right_out[1])
{
  double left = compute(left_in[0], center_in[0],
                        center_in[1]);
  left_out[0] = left; center_out[0] = left;

  for(int i = 1; i < block_size; i++)
    center_out[i] = compute(center_in[i - 1], center_in[i],
                            center_in[i+1]);

  double right = compute(center_in[block_size - 2],
                         center_in[block_size - 1], right_in[0]);
  right_out[0] = right; center_out[block_size - 1] = right;
}
```

(b) OpenStream task computing `block_size` elements of the 1d stencil computation per task.

Figure 1: An example of the horizontal task fusion in OpenStream. The presented task computes 1d stencil, where the element i is a function of itself and elements $i-1$ and $i+1$. In this example the fusion is optimal, i.e., the body of the task has no redundant computations and streams were coalesced.

plications into tasks of the right size. Tasks, if too large, reduce available parallelism and, if too small, incur a significant runtime overhead. To mitigate those problems, automated task transformations that can adjust their granularity are needed.

Recently, Nobre et al. [NDRP19, NDRP20] proposed analyses and tiling techniques for OpenStream applications using the polyhedral model. Although this is a first step towards an automated framework, many problems still remain unsolved, such as transformations of non-affine programs, stream coalescing (i.e., merging multiple streams into one), and profitability heuristics. Some of those problems were addressed in the context of other task-parallel programming models [NSZS13]. However, because of its stream semantics, OpenStream presents different challenges, making it troublesome to implement such solutions.

In this poster we give an overview of task fusion in OpenStream, outlining the components that are needed to create an automated framework.

2 Motivation

To quantify the impact of task size on the execution time and motivate the need for task fusion we evaluate the *jacobi-1d* benchmark from the OpenStream benchmarks suite. The main task of the application is based on the stencil calculation that can be implemented similarly to the code on Figure 1b.

The initial input array of the program is partitioned into blocks of size `block_size`, with each block being communicated using its own data stream. The boundary elements are passed around using additional streams (left and right). In this case, task fusion is trivial, as the size of the block can be directly controlled by the user as an input parameter. However, this is not always the case, as the task can be written in many different ways, making more general task fusion a non-trivial problem.

The result of running the benchmark with different block sizes can be seen on Figure 2. Initially the execution time decreases as larger tasks reduce the runtime overhead, however block sizes larger than 2^{14} reduce the available parallelism, harming performance. This ad-

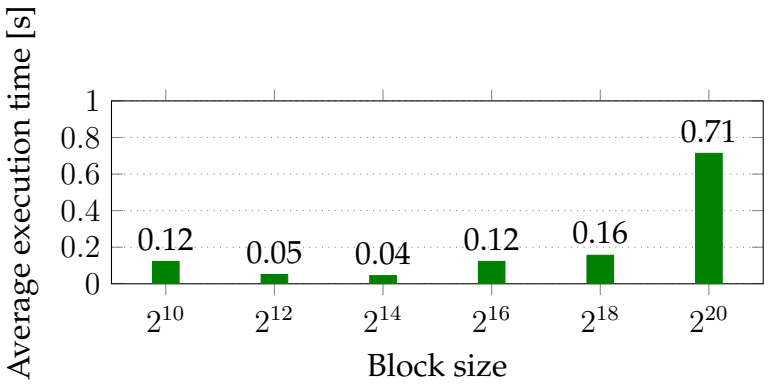

Figure 2: Average execution time of the *jacobi-1d* OpenStream benchmark for a matrix with 2^{20} elements and 128 iterations, executing on an Intel Core i7-6700 processor, 4C/8T, running at 3.40 GHz with 16 GiB of main memory.

vocates the importance of selecting the right task granularity.

3 Task Fusion

We differentiate two types of tasks fusion. *Horizontal* task fusion, when multiple instances of the same task are merged together, or *vertical* task fusion, when two or more instances of distinct tasks types, that communicate directly with each other, are merged.

We showed the importance of horizontal task fusion in Section 2. Now we discuss the problem in more detail by looking at the 1d stencil operation in Figure 1. The naive version of the code, where each element of the array is processed by a separate task, can be seen on Figure 1a. We aim to transform the code into something similar to the task in Figure 1b. Going from one version to the other involves four main steps:

1. *Profitability analysis* - based on static or runtime analyses of the application, e.g., analysis of pressure and back-pressure between tasks, or runtime profiling, that are used to decide whether the fusion is profitable and how many tasks should be merged.

2. *Merging work functions* - this transformation increases the work done by the task. This may be as simple as replicating the work function the required number of times, ensuring the resulting code is correct. Another solution may involve calling the work function repeatedly, by a loop within the task. Stream accesses remain unchanged and the fused task accesses all the streams accessed by its parts.

3. *Task body optimizations* - the result of code fusion most likely results in sub-optimal code with many redundancies. Applying standard compiler optimizations can simplify the code and improve the task's performance.

4. *Stream accesses coalescing* - since streams management is source of overhead in the application, reducing the number of streams used by the task may improve performance significantly. In the context of the provided example, all elements within the block can be communicated using a single stream, instead of keeping a separate stream for each array element. Because of the dynamic nature of streams this is a non-trivial effort, involving data-flow analysis on streams.

The problem of task tiling was partially solved for affine programs [NDRP19], but optimizing programs with arbitrary control flow still remains a challenge.

Vertical fusion follows a similar pattern, however there is no need for stream coalescing, as input and output streams remain unchanged. Instead, the stream communication between merged tasks has to be replaced with communication using buffers local to the task. In fact subsequent optimizations may remove the need for extra buffers by removing boundaries of merged tasks.

4 Conclusion

In this poster we outlined the components needed to develop an automated framework for task fusion, alongside the example motivating this work. In future work, we will develop the techniques discussed here and implement them within the OpenStream compiler and runtime. We believe this work will not only improve the execution of OpenStream applications on the CPU, but also build the foundation for more efficient heterogeneous execution, e.g., by transforming tasks so they can be compiled/synthesized into more efficient accelerators.

References

[CZSS11] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-Java: the new adventures of old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, 2011.

[DAB+11] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel processing letters*, 21(02), 2011.

[DPH+16] Andi Drebes, Antoniu Pop, Karine Heydemann, Albert Cohen, and Nathalie Drach. Scalable task parallelism for NUMA: A uniform abstraction for coordinated scheduling and memory management. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, 2016.

[NDRP19] Nuno Nobre, Andi Drebes, Graham Riley, and Antoniu Pop. Beyond polyhedral analysis of OpenStream programs. In *IMPACT 2019-9th International Workshop on Polyhedral Compilation Techniques*, 2019.

[NDRP20] Nuno Nobre, Andi Drebes, Graham Riley, and Antoniu Pop. Bounded stream scheduling in polyhedral OpenStream. In *IMPACT 2020-10th International Workshop on Polyhedral Compilation Techniques*, 2020.

[NSZS13] V Krishna Nandivada, Jun Shirako, Jisheng Zhao, and Vivek Sarkar. A transformation framework for optimizing task-parallel programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 35(1), 2013.

[PC13] Antoniu Pop and Albert Cohen. OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4), 2013.