Combining Fusion Optimizations and Piecewise Execution of Nested Data-Parallel Programs

Wolf Pfannenstiel

Technische Universität Berlin
wolfp@cs.tu-berlin.de

Abstract. Nested data-parallel programs often have large memory requirements due to their high degree of parallelism. Piecewise execution is an implementation technique used to minimize the space needed. In this paper, we present a combinination of piecewise execution and loop-fusion techniques. Both a formal framework and the execution model based on threads are presented. We give some experimental results, which demonstrate the good performance in memory consumption and execution time.

1 Introduction

Nested data-parallelism is a generalization of flat data-parallelism that allows arbitrary nesting of both aggregate data types and parallel computations. Nested data-parallel languages allow the expression of large amounts of data parallelism. The flattening transformation introduced by Blelloch & Sabot [2] exposes the maximum possible data parallelism of nested parallel programs. As parallelism is expressed by computations on vectors, the memory requirements of flattened programs are proportional to the degree of parallelism. Most implementations, e.g. NESL [1], use libraries of vector operations which are targeted by the compiler. While this approach encapsulates machine-specific details in the library and facilitates code generation, it has at least two drawbacks [4]. First, it further increases memory consumption by introducing many temporary vectors. Second, code-optimizations cannot be applied across library functions. Thus, high memory consumption is one of the most serious problems of nested data-parallel languages.

Palmer, Prins, Chatterjee & Faith [5] introduced an implementation technique known as *Piecewise Execution*. Here, the vector operations work on vector pieces of constant size only. In this way, low memory bounds for a certain class of programs are achieved, but the management of the pieces requires an interpreter.

To tackle both drawbacks of the library approach, Keller & Chakravarty [4] have proposed to abandon the library and to use a new intermediate compiler language $\mathcal{L}_{DT}$ instead, whose main feature is the separation of local computations from global operations (communication and synchronization) using the idea of *Distributed Types*. Local computations can be optimized, the most important optimization being the fusion of consecutive loops.

In this paper, we propose combining piecewise execution and loop-fusion into a single framework. We extend the compiler language $\mathcal{L}_{DT}$ by *Piecewise Types*

J. Rolim et al. (Eds.): IPDPS 2000 Workshops, LNCS 1800, pp. 324-331, 2000.

All optimizations possible in $\mathcal{L}_{DT}$ can still be applied to our extended language. Our implementation uses a self-scheduled thread model, which was introduced in [6].

The rest of the paper is organized as follows. Sect. 2 gives a brief summary of related work. Sect. 3 describes the combination of fusion and piecewise execution. The multi-threaded execution model is looked at briefly in Sect. 4 and we present results of some experiments. Finally, Sect. 5 gives an overview of future work.

2 Related Work

Our work was inspired by two key ideas: *Piecewise Execution*, described in [5], and *Distributed Types*, as proposed in [3].

Piecewise Execution. Piecewise execution is based on the observation that many nested data-parallel programs contain pairs of generators and accumulators. Generators produce vectors that are much larger than their arguments. Accumulators return results that are smaller than their input. The NESL programs in Fig. 1 are examples of matching generator/accumulator pairs. The operation `[s:e]` enumerates all integers from `s` to `e`, `plus_scan` calculates all prefix sums of a vector, `{x * x : x in b}` denotes the elementwise squaring of of `b`, and `sum` adds up all vector elements in parallel. `SumSq(1,n)` returns the sum of all squares from 1 to n. The excess parallelism must be sequentialized to

```
function SumSq (s,e) =
  let
    a = [s:e];
    c = {x * x : x in a}
  in
    sum(c);
```

```
function SumSqScan (s,e) =
  let
    a = [s:e];
    b = plus_scan (a);
    c = {x * x : x in b}
  in
    sum(c);
```

Fig. 1: Example programs in NESL

match the size of the parallel machine. However, if the generator executes in one go, it produces a vector whose size is proportional to the degree of parallelism. Then, memory consumption may be so high that the program cannot execute.

In piecewise execution, the computation exposes a consumer/producer pattern. Each operation receives only a piece, of constant size, of its arguments at a time. The operation consumes the piece to produce (a part of) its result. After the current piece is completely consumed, the next piece is requested from the producer. Once a full piece of output has been produced, it is passed to the consumer as input. Large vectors never exist in their full length, so piecewise execution enables larger inputs to be handled. However, some means of control are needed to keep track of the computation. In [5], an interpreter is employed to manage control flow. Piecewise execution is a tradeoff between space and time.

The memory consumption can be reduced dramatically, while the computation times are likely to increase owing to the overhead associated with serializing flattened programs. Indeed, interpretation involves a significant overhead.

Limits of Piecewise Execution. Some operations are not well suited for piecewise execution, e.g. `permute`. Potentially the complete data vector must be known to produce the first part of the result, because the first index may refer to the last element of the input. Whenever one of these operations occurs in a program, piecewise execution is not possible without buffering more than one piece at a time. A more detailed discussion of limitations can be found in [5].

Distributed Types. In the library-approach, the compilation of nested data-parallel programs is finished after the flattening transformation. The parallel work is delegated to a set of library functions. Optimizations across functions are not possible owing to the rigid interfaces of the library. Keller & Chakravarty [4] decompose library functions into two fractions. First, computations with purely local meaning are extracted, i.e. sequential code that references only local memory. Second, all other computations, such as interprocessor communication or synchronization, are declared as global operations. Adjacent local computations build a block of sequential code to which processor-local code optimizations can be applied. Blocks of global operations can be optimized as well, e.g. two send operations may be combined to form just one.

The intermediate language $\mathcal{L}_{DT}$ is introduced featuring *Distributed Types*, a class of types used to distinguish local from global values. A local value consists of one value per processor and is denoted by $\langle\langle . \rangle\rangle$. The components have only local meaning – their layout and size are known only on the local processor. The function $split\_scalar : \alpha \to \langle\langle \alpha \rangle\rangle$ transforms a scalar of type α into a local value whose components all have the specified value (e.g. by broadcasting it). The function $split\_agg : [\alpha] \to \langle\langle [\alpha] \rangle\rangle$ takes a global vector and splits it into chunks. To transform a local vector into a global one, $join\_agg : \langle\langle [\alpha] \rangle\rangle \to [\alpha]$ is used. The higher-order operation $\langle\langle . \rangle\rangle$ takes a function f and applies it to all components of a local value. To transform a program, the function boundaries of library operations are removed by decomposing and inlining their code. All local computations are represented by instances of two canonical higher-order functions.

1. **Loop** : $(\alpha_1 \times \alpha_2 \to \beta) \times (\alpha_1 \times \alpha_2 \to \alpha_2) \times (\alpha_1 \times \alpha_2 \to Bool) \to [\alpha_1] \times \alpha_2 \to [\beta] \times \alpha_2$
2. **Gen** : $(\alpha_2 \to \beta) \times (\alpha_2 \to \alpha_2) \times (\alpha_2 \to Bool) \to Int \times \alpha_2 \to [\beta] \times \alpha_2$

Loop represents calculations on vectors. It receives a computation function, an accumulating and a filter function as input. The filter can be used to restrict the output to elements for which the function returns $True$. **Gen** is similar to **Loop** except that it loops over an integer rather than a vector, i.e. it can be used to generate vectors. The *split* and *join* operations are used to embed the values into the distributed types. The interface of the vector operations remains the same. When two consecutive operations are split up, often the final *join* of the

first operation and the initial *split* of the second one can be eliminated, leaving larger blocks of local computations.

Fusion Optimizations. Deforestation is a well-known technique for fusing sequential computations on aggregate data structures. As local computation-blocks in $\mathcal{L}_{DT}$ form sequential code, these techniques can be applied here, too. A number of transformations to fuse adjacent **Loop** and **Gen** constructs are presented in [3]. The main benefits are fewer vector traversals and fewer (or maybe no) intermediate vectors. Consider the function `SumSq` given in Fig. 1. In the library approach, three functions are used to implement the program, namely `enumerate` (corresponding to `[s:e]`), `vector_mult` (elementwise multiplication) and `sum`. Transforming the functions into $\mathcal{L}_{DT}$, three blocks of code are formed. In the first part, parameters are split into local values. The last part is the global reduction of all partial sums. All computations in between, which realize the computations inside the three vector operations, form one local block and can be fused into a single **Gen** construct. Thus, fusion combines generation and reduction of the vectors such that no vector is actually created, rendering piecewise execution superfluous here.

Limits of Fusion. **Loop** and **Gen** constructs cannot be fused across global operations, because the purely local semantics of argument values is destroyed by the global operation. Consider `SumSqScan` in Fig. 1. Here, `plus_scan` is split into three parts, the middle part being a global operation originating from `plus_scan`, which propagates partial sums across processors. Unlike in the previous example, not all local code blocks can be fused, because the propagation forms a barrier between the blocks. Whenever a global operation occurs between pairs of **Loop** or **Gen** constructs, fusion is not possible.

3 Combining Fusion and Piecewise Execution

To combine the benefits of piecewise execution and fusion, we extend $\mathcal{L}_{DT}$ by *Piecewise Types*. We call the extended language $\mathcal{L}_{PW}$. It still contains all features of $\mathcal{L}_{DT}$, as we want to support all its optimizations. We use $\lessdot . \gtrdot$ to denote the type constructor for piecewise types. We provide a higher-order function $\lessdot . \gtrdot$, which embeds a computation into a piecewise execution context. If we have $f : \alpha_1 \times \ldots \times \alpha_n \to \beta_1 \times \ldots \times \beta_m$, then $\lessdot f \gtrdot$ has type $\lessdot \alpha_1 \gtrdot \times \ldots \times \lessdot \alpha_n \gtrdot \to \lessdot \beta_1 \gtrdot \times \ldots \times \lessdot \beta_m \gtrdot$. This means $\lessdot f \gtrdot (v_1, \ldots, v_n)$ denotes the piecewise execution of f, where all the arguments v_i must be of a piecewise type. To make a value ready for piecewise execution, we supply a function $pw\_in : \alpha \to \lessdot \alpha \gtrdot$. As only certain parts of programs are to be executed in a piecewise manner we provide $pw\_out : \lessdot \alpha \gtrdot \to \alpha$, which transforms a value of piecewise type into a value of its original type.

Fig. 2 (left) shows an abstract and already fused definition of `SumSqScan`. (The definitions of the instance functions f, g, k, f', g' and k' are not given

since they are not needed for understanding the transformation.) The **Gen** construct implements the `enumerate` generator plus the first local part of `plus_scan` (formerly a **Loop**). The function *propagate_+* is the global operation that hinders full fusion. The **Loop** construct realizes the second part of `plus_scan`, the elementwise squaring plus the local part of `sum`. Finally, the local values are combined to give the global result by *join_+*, which globally sums up all values. The value *accg* is the initial accumulating value needed for **Gen**. (Its definition is omitted for simplicity's sake.)

$$
\begin{array}{l}
\mathit{function}\ \mathit{SumSqScan}'(sg, eg) = \\
\quad \textbf{let} \\
\quad\quad n \quad = \mathit{split\_scalar}(sg - eg) \\
\quad\quad \mathit{acc} \quad = \mathit{split\_scalar}(\mathit{accg}) \\
\quad\quad (v, b) = \langle\!\langle \textbf{Gen}\ (f, g, k) \rangle\!\rangle (n, \mathit{acc}) \\
\quad\quad c \quad = \mathit{propagate\_+}(b) \\
\quad\quad d \quad = \langle\!\langle \pi_2 \circ \textbf{Loop}\ (f', g', k') \rangle\!\rangle (v, c) \\
\quad \textbf{in} \\
\quad \mathit{join\_+}(d)
\end{array}
$$

$$
\begin{array}{l}
\mathit{function}\ \mathit{SumSqScan}'_{pw}(sg, eg) = \\
\quad \textbf{let} \\
\quad\quad n \quad = \mathit{pw\_in}(sg - eg) \\
\quad\quad \mathit{np} \quad = \lessdot \mathit{split\_scalar} \gtrdot (n) \\
\quad\quad \mathit{acc} \quad = \mathit{pw\_in}(\mathit{accg}) \\
\quad\quad \mathit{accp} \quad = \lessdot \mathit{split\_scalar} \gtrdot (\mathit{acc}) \\
\quad\quad (vp, bp) = \lessdot \langle\!\langle \textbf{Gen}\ (f, g, k) \rangle\!\rangle \gtrdot (np, \mathit{accp}) \\
\quad\quad cp \quad = \lessdot \mathit{propagate\_+} \gtrdot (bp) \\
\quad\quad dp \quad = \lessdot \langle\!\langle \pi_2 \circ \textbf{Loop}\ (f', g', k') \rangle\!\rangle \gtrdot (vp, cp) \\
\quad\quad d \quad = \lessdot \mathit{join\_+} \gtrdot (dp) \\
\quad \textbf{in} \\
\quad \mathit{pw\_out}(d)
\end{array}
$$

Fig. 2: Fused (left) and piecwise (right) versions of `SumSqScan`

To execute the function in a piecewise fashion, we lift the operations to piecewise types, convert the arguments into piecewise values first, and finally transform the piecewise output into ordinary values again. The transformed program in $\mathcal{L}_{PW}$ notation is shown in Fig. 2 (right). The input and output have the same type as in the original version, i.e. the caller is not affected.

The canonical representation of vector computations using **Loop** and **Gen** is well-suited for an automatic analysis of memory-critical program patterns. A generator is always a **Gen** with filter function that is not unconditionally false ($\lambda x.\lambda a.False$). (Furthermore, it must not be followed by a projection of only the accumulating value (π_1).) An accumulator is a **Loop** that has a truely restricting filter function (i.e. not $\lambda x.\lambda a.True$). An automatic analysis and transformation of such program fragments remains to be developed.

4 Implementation and Benchmarks

The piecewise behavior of program fragments is realized by employing threads, which simulate the producer/consumer behavior in a couroutine-like fashion. Viewing the program as a DAG, the sink node calculates the overall result. To do so, it requests a piece of input from the node(s) on which it depends. The demand is propagated until a node is reached that still has input to produce its next piece of output. Initially, only the source nodes hold the program input. Whenever a thread completes a piece of output, it suspends and control switches to the consumer node. If a thread runs out of input before a full piece of output has been built, it restarts (one of) its producers. Control moves up

and down in the DAG self-scheduled until the last node has consumed all its input by producing the last piece of the overall result. A detailed description of the execution model can be found in [7]. To enable piecewise execution, the underlying thread model needs to provide only a processor-local switching protocol including control-flow mechanisms and local data-exchange among threads. The model itself allows only sequential execution as there are no means of communication or synchronization among different processors. However, the model does not pose restrictions on the operations executed inside threads, so e.g. the use of message passing libraries like MPI on distributed memory machines is possible. (On SMP machines, semaphores or signals may be employed to coordinate and synchronize threads.)

We can adopt the library approach by realizing each library function as one thread (on each processor). These threads run synchronously working on the same data-parallel operation. Of course, the code must be enriched by control statements to realize the switching behavior for piecewise execution. Communication and synchronization among processors are realized by means of a message passing library. If consecutive function calls have the same data production/consumption rates, they can be encapsulated into one thread. It is also possible to fuse the original program as far as possible first and then embed the remaining code into threads, attaching the piecewise control-structures.

We transformed a number of examples manually and implemented them on a Cray T3E. The programs are written in C in SPMD style using the StackThreads library [8] to realize our thread model. The StackThreads mechanisms have a purely local semantics. Communication and synchronization among processors is realized using the Cray `shmem` communication library. We implemented combinations of library and fused code with piecewise execution, exploiting the generality of our thread model.

One of the examples implemented is the Line-of-Sight algorithm, which determines all objects that are visible from a specified observation point, given the altitudes of the objects lying in the observer's viewing direction. The altitude of the objects might be given by a height function of X and Y coordinates. The algorithm would then take two points in the 2D plane as the observation point and focus, respectively, and calculate which objects on the connecting line are visible using a specified resolution between points. Defined in this way, the algorithm exhibits the typical generator/accumulator pattern. In Fig. 3, running times are shown for two piecewise program versions `par_pw_norm` and `par_pw_fusion`, the first of which being an adaption of the library version and the latter a fused variant. The performance depends heavily on the piece size chosen. If the output is small, we find a pattern that is very similar to results we observed for other programs (e.g. `SumSqScan`), too: a piece size (per processor) below 500 is so small that the overhead for switching dominates computation times resulting in high execution times. The range between 1K and 3K gives the best running times. The sweet spot at 1K elements corresponds to the size of the second-level cache on the Alpha processors. Above 3K, execution times rise significantly because the vector elements begin stepping on one another in the cache. Beyond 6K,

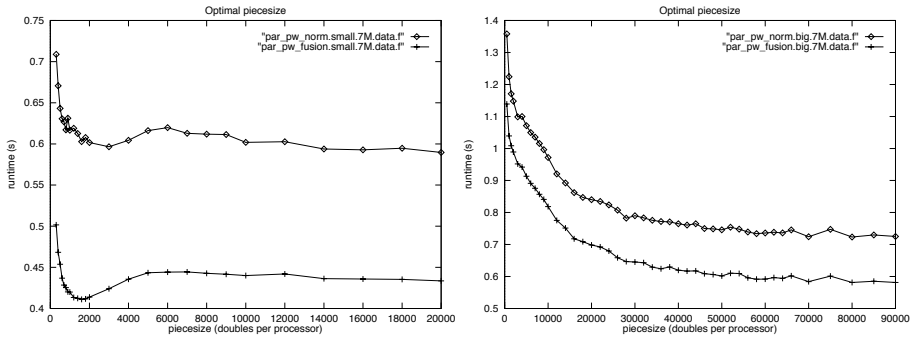

Fig. 3: Optimal piece size for Line-of-Sight with small and large output

the performance improves again slowly owing to the decreased thread overhead. However, the time never again drops below the minimum time attained for small piece sizes. If the output is large, `pw_out` (necessary for assembling the piecewise generated result) has a big communication overhead, which is worse if the piece size is small. This overhead offsets the improved cache usage and dominates the running times of the piecewise program versions. Here, the bigger the piece size, the better the performance. We measured the absolute speedups of four different versions. The piece sizes for the piecewise programs were set to the best values determined in the previously. The results are shown in Fig. 4. The com-

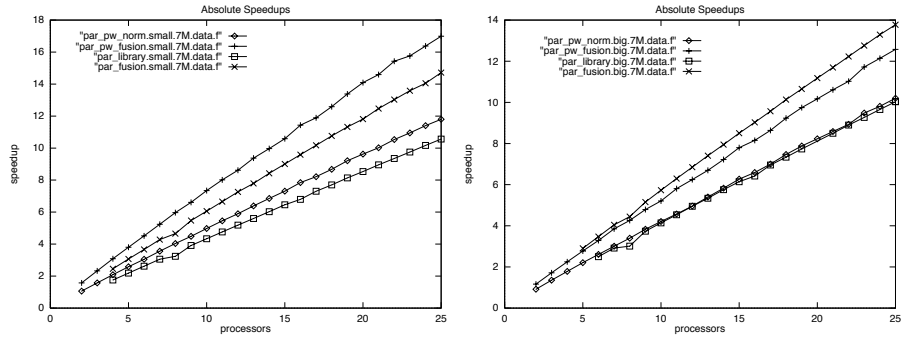

Fig. 4: Absolute speedups of Line-of-Sight with small and large output

bination of fusion and piecewise techniques yields the fastest results if there are few visible points. The fused version (`par_fusion`) is slightly slower. The plain library-approach (`par_library`) achieves the worst runtime. Using piecewise execution with one thread per library function slightly increases the speedups. The optimized use of the cache working on small pieces of the memory would appear to compensate the multithreading overhead. The improvements of fusion and piecewise execution seem to mix well. Both reduce memory requirements in an

orthogonal way. For large output sizes, owing to the communication overhead of `pw_out`, `par_fusion` performs better than `par_pw_fusion`, and `par_pw_norm` is not faster than `par_library` anymore. However, the essential benefit from piecewise execution can be seen in both cases. The piecewise versions can run on any number of processors. For small output, both `par_library` and `par_fusion` need at least four processors to handle 7 million objects. For many output values, `par_library` needs six processors and `par_fusion` five to execute.

The performance results for `SumSqScan` are similar to those for Line-Of-Sight with small output. Piecewise execution combined with fusion gives the best results [6]. Experimental results of further examples and more detailed analysis can be found in [7].

5 Conclusion and Future Work

We have combined piecewise execution with fusion optimization expressed in a special intermediate language. We use an improved implementation technique for piecewise execution based on cost-effective multithreading. Piecewise execution does not necessarily mean increasing runtime. On the contrary, the combination of fusion and piecewise execution resulted in the best performance for typical examples. Piecewise execution allows us to execute a large class of nested data-parallel programs that could not normally run owing to insufficient memory.

We intend to develop transformation rules that automatically find and transform program fragments suitable for piecewise execution to implement them in a compiler.

References

1. G. E. Blelloch. NESL: A nested data-parallel language. Technical report, School of Computer Science, Carnegie Mellon University, 1995.
2. G. E. Blelloch and G. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8(2):119–134, 1990.
3. G. Keller. *Transformation-based Implementation of Nested Data Parallelism for Distributed Memory Machines*. PhD thesis, Technische Universität Berlin, 1999.
4. G. Keller and M. M. T. Chakravarty. On the distributed implementation of aggregate data structures by program transformation. In *HIPS '99*. IEEE CS, 1999.
5. D. Palmer, J. Prins, S. Chatterjee, and R. Faith. Piecewise execution of nested data-parallel programs. In *LCPC '95*. Springer, 1996.
6. W. Pfannenstiel. Piecewise execution of nested parallel programs — a thread-based approach. In P. Amestoy, P. Berger, M. Daydé, I. Duff, V. Frayssé, L. Giraud, and D. Ruiz, editors, *EuroPar'99*, LNCS 1685, pages 445–449. Springer, 1999.
7. W. Pfannenstiel. Thread-based piecewise execution of nested data-parallel programs: Implementation and case studies. Technical Report 99-12, TU Berlin, 1999.
8. K. Taura and A. Yonezawa. Fine-grain multithreading with minimal compiler support - a cost effective approach to implementing efficient multithreading languages. In *PLDI '97*. ACM, 1997.