2010 10th IEEE International Conference on Computer and Information Technology (CIT 2010)

Exploiting More Parallelism from Applications Having Generalized Reductions on GPU Architectures

Xiao-Long Wu Nady Obeid Wen-Mei Hwu

Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
{xiaolong, obeid1, w-hwu} @illinois.edu

Abstract

Reduction is a common component of many applications, but can often be the limiting factor for parallelization. Previous reduction work has focused on detecting reduction idioms and parallelizing the reduction operation by minimizing data communications or exploiting more data locality. While these techniques can be useful, they are mostly limited to simple code structures. In this paper, we propose a method for exploiting more parallelism by isolating the reduction from users of the intermediate results. The other main contribution of our work is enabling the parallelization of more complex reduction codes, including those that involve the use of intermediate reduction results. The proposed transformations are often implemented by programmers in an ad-hoc manner, but to the best of our knowledge no previous work has been proposed to automate these transformations for many-core architectures. We show that the automatic transformations can result in significant speedup compared to the original code using two benchmark applications.

1. Introduction

In recent years, scientific applications like image processing and computational geometric algorithms have enjoyed a tremendous performance improvement on data parallel, many-core architectures such as NVIDIA and AMD GPUs. These applications often use reduction operations for data summation, finding min/max values, true element counting, etc., and though reduction may not dominate the overall application performance, it can prevent large code segments from being parallelized. Previous works focused mostly on minimizing the data communication caused by reductions [5,6,10,12,13,15], by inserting directives in the original code to reduce data dependencies among workers and to minimize data communication. These techniques complicate the code and generate a lot of branches, which makes them less suitable for automatic code transformation for wide SIMD architectures such as GPUs. Our focus, in contrast, is on exploiting more parallelism from the original code by isolating the dependencies introduced by the reductions. Our main contributions are as follows:

- The proposed transformation can help exploit more parallelism from scalar and aggregate reductions by isolating loop-carried dependencies.
- It provides an innovative code transformation which is practical for reductions inside complex program structures and those with dependencies in between.
- The proposed transformations can better map to GPUs than previous automatic approaches.

The remainder of this article is organized as follows. Section 2 lists the definition and taxonomy of reductions. Section 3 reviews previous work. Section 4 specifies the conditions used to detect reduction. Section 5 describes our transformation methodology in detail. Section 6 lists the experimental results and Section 7 concludes.

2. Definition and Taxonomy of Reductions

In this work, we define *Reduction* as a repetitive operation performed on a set of values, known as *reduction factors*, to produce one or multiple final results.

The most basic reduction form produces a single reduction result. This reduction form is called *scalar reduction* [2] or *single address reduction* [13]. Such a reduction is illustrated by Figure 1. *S* is the *reduction variable* that stores the reduction result. *A* is the *reduction factor* which can be a constant, a variable, a value-generating function, or an array. *U* is the *consumer of intermediate results*. The *reduction operator* can be an intrinsic *C* operator, or an overloaded operator in *C++*.

The reduction variable can also be a vector of multiple reduction results, as shown in Figure 2. We call such a reduction form an *aggregate reduction*. *A1* and *A2*, which may or may not be the same, are called the *first* and *second reduction factors*, respectively.

978-0-7695-4108-2/10 $26.00 © 2010 IEEE
DOI 10.1109/CIT.2010.213

```
for (i = 0; i < N; i++) {
    S = S + A(i);
    U(S);
}
```

Figure 1. A scalar reduction

```
for (i = 0; i < N; i++) {
    S[A1(i)] = S[A1(i)] + A2(i);
}
```

Figure 2. An aggregate reduction

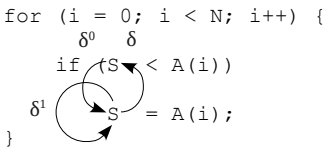

Figure 3. A max reduction

In this work, we classify min/max reductions [2], which find the minimum or maximum value in an array, as complex reductions. We define them as such because they contain conditional statements as shown in Figure 3. We also define reductions with user-defined/overloaded operators as complex reductions because the compiler may not understand their behavior the same way it can understand intrinsic operators, such as addition and multiplication.

3. Related Work

The recognition of scalar, aggregate, and min/max reductions has been well studied since the 1990's [1,2,9,10, 13,14,15]. A reduction statement can be detected by the pattern-matching of dependence graphs, as shown by the arcs in Figures 3, 4, and 5. However, none of the previous techniques consider the use of partial reduction results inside the loop, and only discuss reductions with intrinsic operators. In our work, we relax these two restrictions.

The previous work on reduction parallelization has mostly focused on either minimizing the data communication over reduction variables and factors, or exploiting the data locality on reduction factors at compile-time [5,8,10, 13,15] or run-time [6,7,12]. In [10,13], the transformation focused on parallel execution through Message Passing Interface (MPI) or compiler-supported directives. The reduction variable is privatized for each processor to reduce data communication. Figure 6 shows an example where processors 0, 1, and 2 create a private version of the reduction variable and compute a partial result into it. Afterward, the partial results from each processor are combined. Plata et al.[12] study histogramming on a shared-memory multiprocessor platform. First, the histogram reduction statement is recognized using the techniques in [13, 14], then optimized for data locality of reduction factors at run-time. The problem is modeled as a vertex coloring problem to maximize the inter-iteration write locality. Afterward, loop

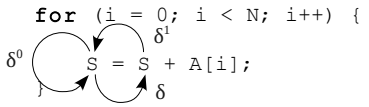

Figure 4. Code pattern for a scalar reduction

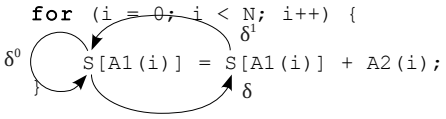

Figure 5. Code pattern for an aggregate reduction

iterations with the same colors are executed in parallel. Han et al. [7] introduce the Local-Write technique to parallelize aggregate reductions by partitioning the loop iterations at run-time to keep the data accesses local and avoid replicating the whole reduction vector, at the cost of increasing data communication between processors.

All the previous techniques have targeted traditional multiprocessor architectures. For such architectures, with relatively few scalar cores, coarse-grained parallelism is sufficient. As Figure 6 shows, every core can reduce a subset of all the reduction factors into a partial result before adding its result to those of all the other cores. Small amounts of load imbalance are tolerable in multi-core architectures. Furthermore, in most cases, replicating the reduction variable to minimize communication does not incur a large memory overhead.

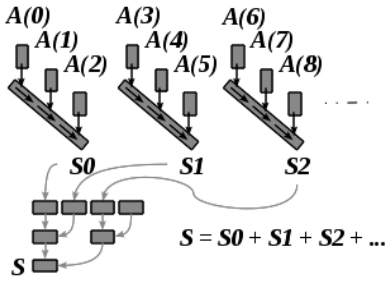

Figure 6. Graphical representation of privatized reduction variables

However, the same transformations cannot be directly applied to many-core architectures like GPUs. GPUs achieve the best performance by executing thousands of light-weight threads simultaneously, therefore fine grained work distribution is preferred over coarse-grained partitioning. Moreover, load imbalance among threads is much less tolerable on many-core architectures, which are more similar to vector machines. Finally, the memory footprint scales with the exploited parallelism and the size of the final result, due to variable replication. This is particularly bad for GPUs and aggregate reductions. For all these reasons, we propose a different transformation which is more suitable for many-core architectures.

```
for (i = 0; i < N; i++) {
  if (…) { S = S + A1(i); }
  else { S = S + A2(i); }
}
```

Figure 7. Code pattern for reduction with a scalar reduction result

4. Detecting Reductions for Transformation

In order to recognize candidates for our transformation, we define several conditions that can be verified at compile-time. The first and most essential is that the reduction operator must be associative and commutative, in other words, the order in which the operations are performed should not affect the correctness of the result. Furthermore, as shown by the arcs in Figure 4, self flow dependence, output dependence, and anti-dependence on the reduction variable must exist in order to use the reduction recognition techniques presented in [1,13,14]. The two conditions above are necessary for detecting parallelizable reductions. The next four are more specific to determining a candidate for our transformation. First, the loop bound of the loop containing the reduction statement must be a loop invariant, since the loop bound will determine the number of GPU threads. This also means that the loop cannot terminate early using a `break`, `goto`, `exit`, or `return` statement. Second, the reduction factor(s) cannot have direct or indirect dependence on the reduction variable. However, any outer loops that contain the reduction loop and that do not contain loop-carried dependencies can be parallelized. The Mandelbrot benchmark in Section 6 is an example of this. Third, loop bodies containing multiple paths should have a reduction statement to the same variable defined along every path; otherwise, an identity value should exist for the reduction operator (e.g, 0 for addition, 1 for multiplication). This condition allows the transformation of codes like the one shown in Figure 7, because every thread is still generating exactly one reduction factor. Finally, for aggregate reductions, the indices of the source and destination variables must be the same, otherwise the code no longer corresponds to any of the reduction types described in Section 2.

5. Transformation

In this section, we will describe the proposed transformations of scalar and aggregate reductions. The transformations are first described for simple scalar and aggregate program structures. We then address more complex structures.

5.1 Transformation of Simple Reductions

The code on the left hand side of Figure 9 is an example of a reduction that is recognized as a candidate for transformation based on the conditions described in Section 4. The code on the right-hand side is the code generated after the transformation. The steps involved in the transformation are the following:

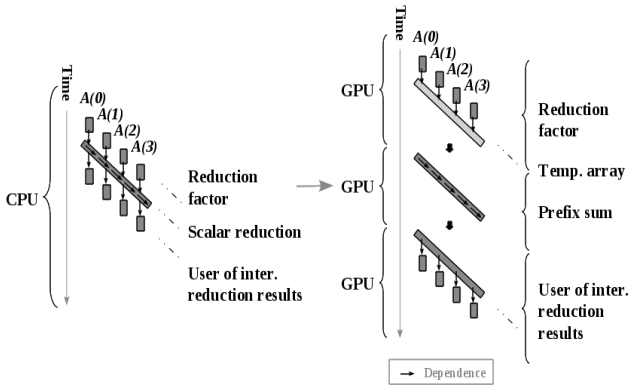

Figure 8. Graphical representation of the transformation

Step 1 is to generate the reduction factors. In this step, all the reduction factors are calculated and stored in a temporary array of size *N*, where *N* is the original loop bound. The parallelized step 1 typically contributes the majority of the performance improvement from the transformation. Step 2, a prefix sum is performed on the factor array using the same operator used in the original reduction. The result of the prefix sum operation is a partial reduction result for every array element *i* of all the elements from 0 to *i* [3]. The result at index *N* is the final reduction result, which is assigned to the final reduction variable. Step 3 is to pass all the intermediate results to the intermediate result consumer, if any exist. This third step may or may not be automatically parallelizable, based on the dependence pattern of the consumer, but this is beyond the scope of this paper. Note that if there is no intermediate results consumer, it suffices to perform a scalar reduction rather than a prefix sum. Figure 8, summarizes the transformation described in this section.

For aggregate reductions, similar transformations occur. First, both factors are calculated into arrays, as shown in Figure 10. Secondly, a call to a library implementation is made since prefix sum cannot be applied to aggregate reductions. The library function takes as arguments the reduction variable vector, the first and second factor arrays, and the loop bound *N*. Note that generating intermediate results for consumers of aggregate reduction variables is very difficult to do. Fortunately, it is not common practice to use the intermediate results of an aggregate reduction and we were not able to find any real application example that do. For that reason, we do not address this issue in this paper.

5.2 Reduction of Complex Program Structures

In real applications, multiple nested reductions may be present as shown on the left-hand side of Figure 11. If the

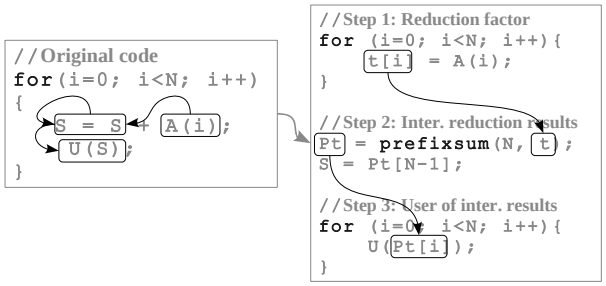

Figure 9. A graphical view of the transformation of scalar reductions

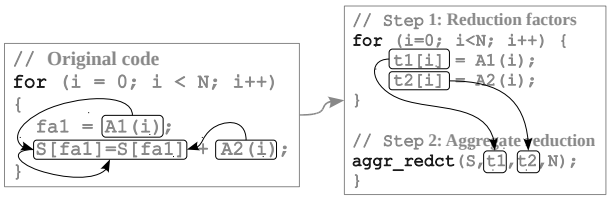

Figure 10. A graphical view of the transformation of aggregate reductions

nested reductions depend on each other, then a simple and systematic extension of the transformation explained in Section 5.1 can be applied. In essence, given two nested reduction variable S1 and S2, where S2 is in the innermost loop which depends on S1, then the transformation can be performed for S1 as explained in Section 5.1, treating the code for S2 as the consumer of the intermediate values of S1. Then in a second iteration, we repeat the transformation process for S2. This iterative approach can be applied to any number of nested reductions until there are no more transformations left to perform. However, if there are no dependencies between the variables, as is the case in the code in Figure 11, this approach is inefficient as it breaks the code into too many small loops. Instead, our transformation method can transform the code in a much more efficient way, as shown in Figure 11. Step 1 now populates all three factor arrays, in a similar code structure to that of the original code. Then in step 2, three independent reductions are performed for each iteration variable. Except for the most complex cases, step 1 can typically be parallelized as a single kernel.

5.3 Automating the Transformation

All of the transformations described in this section can be automated. Our compiler can currently detect scalar, aggregate and complex reductions. Furthermore, the conditions described in Section 4 are simple to detect in a compiler, in order to generate the new source code as shown in sections 5.1 and 5.2. The automatic transformation of 'Step 1', 'Step 2', and 'Step 3' codes into their parallel kernel equivalents is still under development.

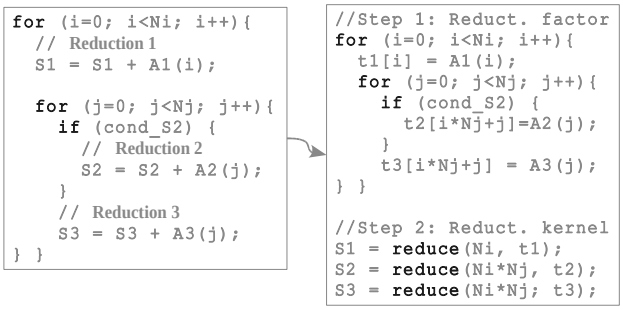

Figure 11. Multiple reductions without dependencies

Though the work presented in this paper does not encompass all the transformation stages from a running serial code to a running parallel code, it contributes a significant step. The first contribution is that of making the code more regular for further automatic transformations. This regularity also makes it easier for the programmer to hand-tune the code. The second contribution is exposing some parallelism even when the original loop is not entirely parallelizable. For example, even if the generation of the reduction factors is not parallelizable, by transforming the prefix sum and the use of intermediate results into parallel kernels, we can still benefit from some parallelism. Finally, as with all compiler optimizations, the automatically generated code is less error-prone for complex program structures, and minimizes the programmer's effort to manually perform these transformations.

Our framework currently applies the same general transformations to all reductions, and does not optimize for every application individually. Because of that, it may sometimes fail to parallelize a piece of code that a programmer can. One example is when the loop bound or product of nested loop bounds causes the temporary array to be larger than the size of the device memory. This situation fails in the automatic framework since we currently favor maximum expansion of the data and minimum work per thread. On the other hand, the programmer can better evaluate the trade-off between data expansion and thread workload to achieve the best transformation that does not exceed available resources. Integrating this trade-off analysis into the compiler will be part of future work.

6. Experimental Results

In this section we demonstrate the speedup achieved by our transformation compared to the sequential code of two benchmarks. Since the stages following our transformation are not yet automated, we manually port the transformed code to the GPU without any hand optimizations. In other words, we simply replace the *for* loop construct with its kernel equivalent, replacing every iteration with a thread. Therefore, the performance we present is for the wor-

Table 1. Experimental results for TPACF

	Seq. (ms)	Hand Coded (ms)	Automatic (ms)	Speedup (H./A.)	Speedup (S./A.)
GPU Kernel	0.0000	1.0717	14.4051	0.07x	-
Data Transfer	0.0000	0.0851	7.4724	0.01x	-
CPU Runtime	76.4048	0.0107	0.0022	4.89x	-
Total	76.40	1.17	21.88	0.05x	3.49x

Table 2. Experimental results for Mandelbrot Set

NPOINTS	Seq. (ms)	Hand-coded (ms)	Automatic (ms)	Speedup (H./A.)	Speedup (S./A.)
100	92.09	246.10	322.64	0.76	0.29
500	499.04	246.04	322.57	0.76	1.55
5,000	2,042.68	246.19	323.25	0.76	6.32
10,000	3,878.59	491.15	649.58	0.76	5.97
50,000	19,498.98	1,228.32	1,615.79	0.76	12.07

st-case scenario where no optimizations other than our transformation are performed.

The first benchmark we use is TPACF from the Parboil suite [11]. TPACF is an equation used as a way to measure the probability of finding an astronomical body at a given angular distance from another astronomical body. TPACF is mostly an application of histogramming, which is a simple case of aggregate reduction. Using our approach, we transformed the original code, as described in Section 5.1, and replaced the histogram statement with a call to a parallel histogramming library. The execution run-time can be seen in Table 1 for the CPU code, the GPU code generated by our approach, and the user optimized hand-coded GPU code. The automatically transformed code achieves a 3.49x speedup over the sequential code. Note that the hand-coded parallel code achieves 65x in comparison. The reason for this large difference is that, in addition to being parallel, the hand-coded version uses a different data tile size than the sequential code, one that better fits the GPU. Specifically, the hand-coded version processes all the data in one tile as opposed to the sequential code that break the input data into 100 tiles. On the other hand, the automatic version simply parallelizes the reduction without any re-tiling, therefore causing the GPU execution to be broken down into 100 kernel calls. Because each of the 100 kernels is too small to saturate the GPU, kernel run-time suffers. In addition, the data transfer time increases due to many transfers between the GPU and CPU, versus a single round-trip in the hand-coded version. However, despite a slightly unfair comparison between the sequential and hand-coded parallel version, these numbers illustrate an important point. Due to compiler limitations, our transformations are only capable of porting the code to the GPU without any kernel-specific optimizations. However, most manual transformations are not done that way. A programmer, being far more capable at parallelizing code than a compiler, is able to perform optimizations more specific to the given kernel, and in some cases is able

```
for (i = 0; i < NPOINTS; i++) {
  z.re = points[i].re;
  z.im = points[i].im;
  for (j = 0; j < MAXITER; j++) {
    ztemp = (z.re * z.re) - (z.im * z.im)
            + points[i].re;
    z.im = z.re * z.im * 2 + points[i].im;
    z.re = ztemp;
    if (z.re*z.re + z.im*z.im > THRESOLD){
      outside++;
      break;
} } }
```

Figure 12. Code segment of Mandelbrot set area

```
// Step 1: Reduction factor
for (i = 0; i < NPOINTS; i++) {
  z.re = points[i].re;
  z.im = points[i].im;
  for (j = 0; j < MAXITER; j++) {
    ztemp = (z.re * z.re) - (z.im * z.im) +
            points[i].re;
    z.im = z.re * z.im * 2 + points[i].im;
    z.re = ztemp;
    if (z.re*z.re + z.im*z.im > THRESOLD){
      t[i] += 1;
      break;
} } }
// Step 2: Reduction operation
outside = reduction(NPOINTS,t);
```

Figure 13. Transformed code segment of Mandelbrot Set by our transformation

to make algorithmic changes to the code, to better fit it to GPU execution. Despite that, it is worth noting that in the case of TPACF, the user is able to achieve 3.49x speedup without any effort on their end. In fact, even a user who is not familiar with GPU programming can achieve some significant, though not optimal, speedups with the help of our compiler transformations.

The second benchmark is the Mandelbrot Set code in the OpenMP Source Code Repository (OmpSCR) [4]. Mandelbrot set is defined as a sequence of complex quadratic polynomials in the mathematical form of "$z_{n+1} = z_n^2 + c$", where c is a given complex value. The equation can be used to determine whether a complex value c is bounded or not. In OmpSCR, the set is calculated by Monte Carlo sampling to estimate the set area. The code in Figure 12 determines if every complex value in the array *points* is converging or not. If not, variable "*outside"* is incremented and the loop breaks.

The inner for-loop for Mandelbrot is not parallelizable because it contains loop-carried dependencies, highlighted in Figure 12, as well as a break statement which could terminate the loop early. However, if the reduction statement in the inner-most loop is removed, the outer loop can be parallelized since its loop bound is constant. As shown in Figure 13, the result of the transformation is a new loop where every iteration executes the entire inner *for*-loop, computes a partial sum for the variable "outside" and stores it in a temporary array of size NPOINTS. In the end,

this temporary array is reduced by a parallel reduction code template. For Mandelbrot, the difference between our automated transformation and the hand-coded transformation is simply a set of optimizations such as shared memory usage, and loop unrolling, resulting in the hand-tuned code being 34% faster than the automatic one, as shown in Table 2. Nevertheless, we achieve a significant 12.07x speedup over the sequential code for NPOINTS=50,000, and this without any effort required from the programmer.

7. Conclusions and Future Work

In this paper we presented an innovative transformation that combines both scalar expansion and loop distribution compilation techniques to parallelize a complex reduction code segment. This transformation provides three advantages. One is the isolation of the dependencies inside the reduction code segment to exploit more parallelism from the the rest of the code. The second is improved capability to transform more complex reductions such as histograms. The third is better portability to many-core architecture with SPMD support. The preliminary experimental results show competitive performance compared to the sequential code.

Future work still needs to be done in completing the implementation, and researching the later stages of the compiler in order to achieve a full-pipeline that automates the transformation from working sequential code to work GPU parallel code. Further more, the threshold for which a reduction is more efficient on the GPU versus the CPU is currently unknown, and to the best of our knowledge, has never been studied. Determining the correct heuristic to choose whether to port a certain reduction to the GPU or not will further increase the usability of our compiler framework.

Acknowledgments

The authors acknowledge the high-performance computing resources provided by Institute for Advanced Computing Applications and Technologies (IACAT). This work was conducted on the infrastructure built using NSF grant 0551665.

References

[1] R. Allen and K. Kennedy "Optimizing Compilers for Modern Architectures," *Morgan Kaufmann Publisher*, October 2001.

[2] M. Arenaz, J. Touriño, and R. Doallo, "XARK: An EXtensible Framework for Automatic Recognition of Computational Kernels," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 30, issue 6, 2008.

[3] Mark Harris, Shubhabrata Sengupta, and John D. Owens. "Parallel Prefix Sum (Scan) with CUDA". *GPU Gems 3*, chapter 39, pages 851–876. Addison Wesley, August 2007.

[4] A. J. Dorta, C. Rodriguez, F. de Sande, "The OpenMP Source Code Repository," In *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pp. 244-250, February 2005.

[5] M. Gupta, S. Midkiff, E. Schonberg, P. Sweeney, K. Y. Wang, and M. Burke, "PTRAN II - A Compiler for High Performance Fortran," *4th International Workshop on Compilers for Parallel Computers*, 1993.

[6] E. Gutiérrez, O. Plata, and E. L. Zapata, "Improving Parallel Irregular Reductions Using Partial Array Expansion," In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, 2001.

[7] H. Han and C.-W. Tseng, "Improving Compiler and Run-Time Support for Irregular Reductions Using Local Writes," In *Proceedings of the 11th international Workshop on Languages and Compilers For Parallel Computing*, August 1998.

[8] S. Hiranandani, K. Kennedy, and C.-W. Tseng, "Compiling Fortran D for MIMD Distributed-Memory Machines," *Communications of the ACM*, vol. 35, no. 8, pp. 66-80, August 1992.

[9] S.-w. Liao, "Parallelizing User-Defined and Implicit Reductions Globally on Multiprocessors," *Lecture Notes in Computer Science*, *Springer-Verlag*. Also in *Proceedings of Annual Asia-Pacific Computer Architecture Conference (ACSAC06)*, Shanghai, PRC, September 2006.

[10] B. Lu and J. Mellor-Crummey, "Compiler Optimization of Implicit Reductions for Distributed Memory Multiprocessors," In *Proceedings of the 12th International Parallel Processing Symposium*, 1998.

[11] Parboil Benchmark Suite, http://impact.crhc.illinois.edu/parboil.php

[12] O. Plata, R. Asenjo, E. Gutiérrez, F. Corbera, Angeles Navarro, and Emilio L. Zapata, "On the Parallelization of Irregular and Dynamic Programs," *Parallel Computing*, vol. 31, issue 6, pp. 544-562, June 2005.

[13] B. Pottenger and R. Eigenmann, "Parallelization in the Presence of Generalized Induction and Reduction Variables," *Technical Report 1396, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & Dev*., January 1995.

[14] B. Pottenger and R. Eigenmann, "Idiom Recognition In The Polaris Parallelizing Compiler," In *Proceedings of the 9th International Conference on Supercomputing*, 1995.

[15] T. Suganuma, H. Komatsu, and T. Nakatani, "Detection and Global Optimization of Reduction Operations for Distributed Parallel Machines," in *Proceedings of the 1996 ACM International Conference on Supercomputing*, Philadelphia, PA, May 1996.